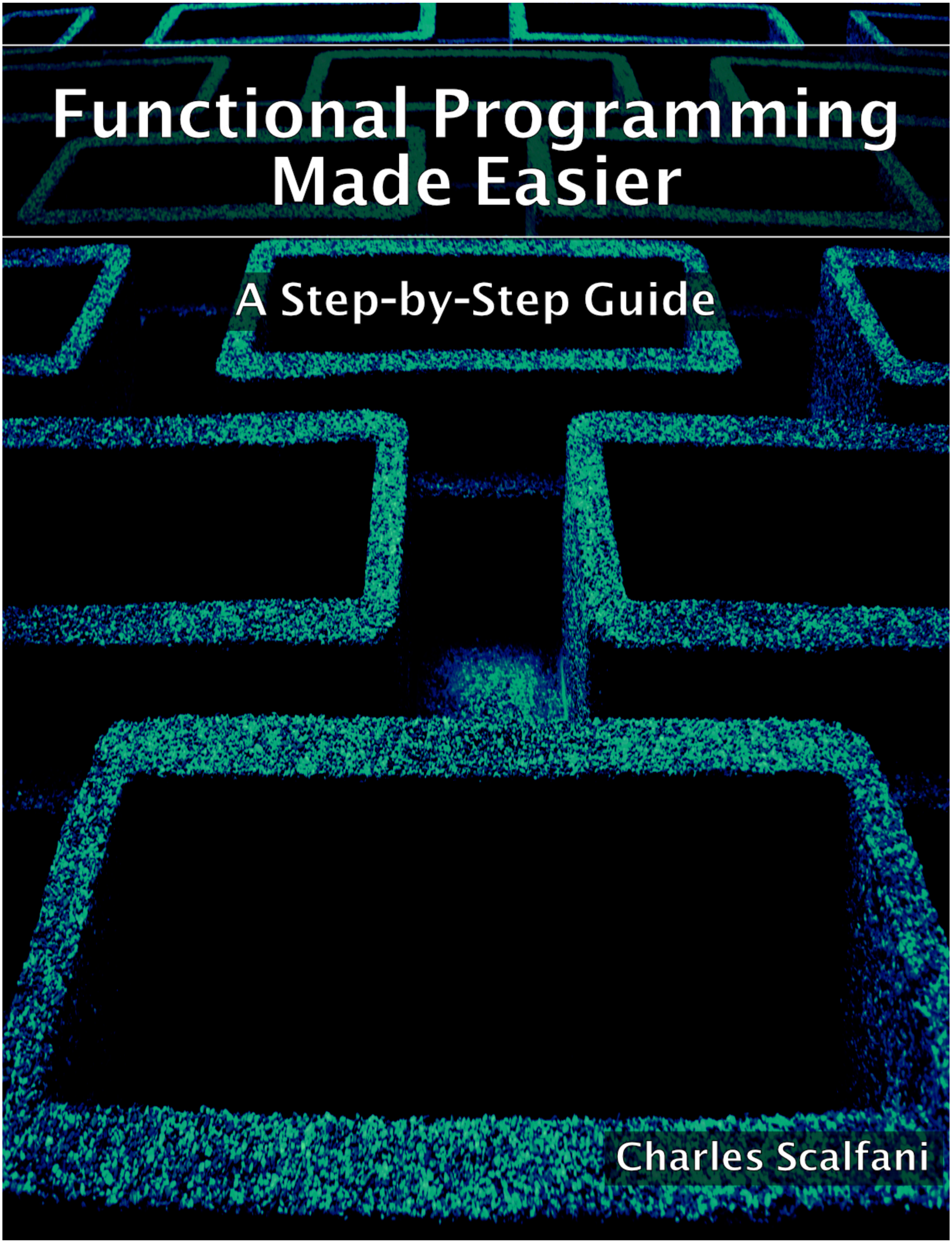
Functional Programming Made Easier

A Step-by-Step Guide

Charles Scalfani



Functional Programming Made Easier

A Step-by-Step Guide

Charles Scalfani

Table of Contents

Introduction

Learning Functional Programming is Hard.

I failed many times to learn Haskell, the Granddaddy of all Functional Programming Languages, before I finally broke through. But not before trying to use Functional Concepts in Javascript. And not before learning Elm (a Purely Functional DSL for web-based, front end GUIs).

It was so hard that I tried anything to avoid the pain. When I tried to learn from free sources on the Internet, I was unable to tell the difference between something I was ready to learn and something I would never need to know as a professional Functional Programmer.

Learning on my own took too long and was too painful. Even though, Functional Programming is more advanced than Imperative Programming, [1] I realized that it didn't have to be that difficult.

The reason that it's so difficult is because there is no definitive guide to turn you from an Everyday Programmer, who programs in Imperative Languages, into a Functional Programmer, who can use Functional Languages like Elm, PureScript, Haskell, etc. to solve everyday problems encountered at work.

This doesn't mean there aren't lots of great blog posts or books. The problem is that there wasn't a single source that would take me by the hand from A to Z [2]. And as a novice, I couldn't tell which books were good and which ones were not. Which blog posts I was ready to read and which ones were far beyond my current reach.

And as a busy CTO of a small company, it was difficult to find the time and energy required to learn Functional Programming. It baffled me that after 35 years of programming experience, that I was struggling to learn yet another language.

And that's the thing. I was not learning just another programming language.

Learning Programming is Learning How to Think

When we learn our first programming language, we also learn how to think like the computer. This way of thinking isn't that foreign to us. Many people use a recipe as an example of a computer program.

First, get out the bread. Then the mustard. Then apply mustard to the bread. And so on.

This way of thinking is exactly how we see our world and so we've been learning how to think that way our whole lives. And this closely matches how the computer operates.

First fetch a value from an address and place it into a Register. Then fetch another value into another Register. Then add them putting the result into yet another Register. And so on.

A sequential, step-by-step process that we're all too familiar with. We still have to learn how to think like the computer since the computer is a pretty simple machine. So complex problems have to be broken down into something a simpleton can do.

But thinking of a program as a sequence of instructions that will be followed by the computer isn't that different from how we accomplish tasks in the real world. So the only real struggle is learning how to map complex problems to a set of basic functions that can be efficiently executed by a computer and yet still understood by a human.

Learning Functional Programming is Learning to Think Differently

Many people find programming to be much easier than Math. To think like a Mathematician is difficult and unnatural. Our natural way of thinking is to jump to unsubstantiated conclusions quickly.

While this probably served us well when we thought we saw a predator on the savannas of Africa, it doesn't always produce accurate results. This is why producing a proper Mathematical Proof is a learned skill that takes many years of practice.

While Programming requires some practice, an unskilled person can produce a working program pretty easily. Professionals may cringe at the sight of their code and it may fail on some edge cases or perform poorly, but, nevertheless, it is a working program.

No such luck as a Mathematician. If you miss an edge case, the whole proof is wrong and unusable. Math holds us to a higher standard and therefore requires us to think differently, i.e. more precisely.

The same is true for Functional Programming. We must think differently and unnaturally, very much like the Mathematician.

However, I want to be very clear here. You do NOT need to be a Mathematician to use Functional Programming but by the end of this book, you will think more Mathematically even if you don't realize you're doing so.

Why Functional Programming?

There have been many attempts to answer this question [3] but I'm going to take a different tact. Instead, I'm going to tell you some of the things I value in programming.

I value:

- Software whose behavior can be easily reasoned about by simply reading the code.
- Programs that don't require extensive testing.
- Powerful abstractions that allow me to get a lot of work done with very little effort.
- Beauty in code and design.
- Reduction or complete elimination of boilerplate code.

- Reusable code.
- Confidence in code correctness.
- Elimination of unnecessary complexity.
- Coding mostly the Happy Path.

I cannot imagine many programmers who don’t value the things on this list. We all want powerful tools that help us solve problems without requiring us to do the same work over and over again while producing code that’s predictable and maintainable.

We don’t want to deploy our code into production only for it to break requiring us to stay up all night trying to understand code that we wrote a few months ago under the pressure of production being down.

Essentially, we want to be in control of our code to minimize pain, and anything that makes our job easier and less painful is valuable to us.

If you value these things, then Functional Programming is the best and most powerful tool that we have to date to achieve everything on this list and more. That’s not to say that something more powerful may not come along in 10 to 20 years, because it could.

But unlike Imperative Programming, Functional Programming has its roots in Mathematics that goes back over 100 years and that Math is built on a foundation of Mathematics that goes back hundreds if not thousands of years. In contrast, conventional programming languages only go back about 60 years.

It is said that Mathematics is discovered not invented. This implies that Functional Programming is also discovered, whereas most other languages are simply invented. [4]

And since Mathematics has proven itself so valuable for solving complex problems in nearly every human endeavor, it’s pretty likely that Functional Programming is our best hope to control the scourge of complexity that’s pervasive in our modern software.

You Cannot Describe an Experience

Ever hear someone say, "You had to be there"? Well, trying to explain why Functional Programming is so great is somewhat like that.

I know now that, when given the choice, I’ll never choose an Imperative Language over a Functional one ever again.

However, if you pressed me for an explanation, I’d say because the *experience* of programming in a Functional Language is just so far superior that I enjoy programming infinitely more.

Unfortunately, this explanation is terrible because I cannot describe an experience well enough so that you will understand, unless, of course, you’ve shared that experience.

It’s like trying to explain to a couple who are about to have their first baby what it’s like to become parents and how the center of their universe will now be on the child. I don’t even try explaining this. Instead, I simply wait until after their child is born and then I can have a conversation with them regarding the ups and downs of parenthood.

Functional Programming is similar. As you work through this book, you will slowly experience what so many other programmers have experienced on their journey. You'll fully understand what I mean by "far superior experience" after you build your first real-world project, which we will do in the last Chapters of the book.

Barrier to Entry

Javascript probably has the lowest Barrier to Entry of all programming languages because it's already installed on nearly every computer inside of every Browser. Even your Grandmother has access to Javascript and, if so inclined, could start writing a line or two of Javascript code today.

You don't have to be a programmer to use Javascript. Many people just copy code from StackOverflow and paste it into their editor without ever understanding how it works. This doesn't always bode well and often they need to hack haphazardly at the code to cajole it to work. With very little skill, people can bootstrap themselves to become Javascript programmers, albeit not very good ones.

The Barrier to Entry for Functional Programming is the highest I've ever encountered. Partly why I wrote this book was to help reduce (not eliminate) the amount of effort required to enter into the Functional landscape.

So far, it may seem like I'm making an argument for why NOT Functional Programming, which would be true if not for one very important thing, Costs of Ownership.

Costs of Ownership

One thing that most people dismiss when buying a new car is the cost of owning that new car. When deciding between two different cars most people simply consider the initial upfront costs, i.e. the sticker price, while completely overlooking the fact that they have very different Costs of Ownership. It is the rare individual who takes this into consideration when they are sitting in the car salesman's office.

Ideally, one would want to know what the hourly cost of labor is at the dealership or what the manufacturer charges for a new alternator or radiator.

Another useful piece of information is reliability. The cost to fix the more expensive car may be higher than the cheaper one, but if it's more reliable, then that's another factor to consider. Repairing a cheap car often will add up quickly.

Other useful information is insurance costs and fuel efficiency.

These are all Costs of Ownership for a car and armed with this knowledge, we could make far better decisions regarding which car to buy.

Case Study: Javascript

It’s no different for software. We developers consider mostly language popularity and job prospects when choosing which language to hang our hat on. However, we rarely consider the Costs of Ownership.

One of the things I really liked about Javascript when I first started using it was how quickly I could write something and get it to work. I was coming from Java and welcomed the freedom from what I considered, at the time, "its tyrannical adherence to Types".

In the early days of Javascript, I’d sometimes get a Runtime Exception when I ran my code but I could easily and quickly fix it. Little did I realize, that this experience would scale up exponentially with the size of my program.

Having a dynamic language felt freeing and more efficient because I didn’t have to wait for it to compile and I could do "anything" to my program and "not have to worry about Types". I could have an Array that contained BOTH Strings and Numbers if I wanted to. In my mind, this was a big win regarding the development phase.

The Cost associated with writing Javascript code was far less than it was with a Statically Typed language, which is why I initially loved it.

But because Javascript isn’t compiled, it routinely crashes in the hands of users. So to combat this, the industry embraced Test Driven Development. It was all the rage and was touted to ferret out bugs and reduce Runtime Errors. Unfortunately, this approach has saddled us with the cost of having to write and maintain test code, which quickly grows to be an appreciable percentage of our codebase with its own set of bugs.

For me, this cost began to outweigh any gains I originally got from using a Dynamic Language.

Case Study: Elm

When our company decided to use Elm instead of Javascript for our next big project, I had to train the developers in basic Functional Concepts which was a one-time, up front cost.

In the beginning, a lot of time was spent trying to decipher the compiler errors. For example, Elm has Type Inference meaning that you don’t always have to specify the Types since the compiler can infer them from how you use your variables. However, this means that the error isn’t always reported where you made your mistake.

It takes developers time to realize that they used a variable differently in two places, e.g. as a String and a Number, and that the second usage is where the error will show up. Sometimes that isn’t always the wrong usage. Maybe it’s supposed to be a Number and the usage as a String is where the real error is.

Even after this initial developer acclimation period is over, an appreciable amount of time is spent dealing with compiler error messages, i.e. mistakes made by the developer that the compiler catches. These bugs are not caught by testing code or by users, but, instead, by the compiler, code that we do not have to write specifically for our program and code that we do not have to maintain.

When we put our Elm code into production, it just seemed to work with no Runtime Errors. [5]

The Upfront Cost of Elm was huge compared to Javascript, but the Costs of Ownership were drastically reduced so much so that we are able to add complex features to our product faster than ever before.

It’s worth noting that this gain wasn’t solely because of the difference between a Statically Typed language and a Dynamically Typed one. Many of the gains in development came from the developer’s ability to more easily reason about their code and to be able to leverage powerful abstractions.

Finding Developers

If you are involved in the hiring process and you’re reading this wondering how you’ll find trained Functional Programmers if you adopt a Functional Language at work, the answer, at the time of this writing, is that you’re probably not, at least not in your area.

If your company is open to remote workers then finding developers is not a problem. There are plenty of Developers who would gladly leave their current employment to get a chance to work in a Functional Programming Language. I saw this personally when we were hiring Elm programmers.

Also, in my experience, Functional Programmers are a cut above most. They’ve typically had to endure the hardship of learning this craft on their own time and are usually lifelong learners. Hopefully, this book will go a long way to reduce that hardship, but it will never be eliminated completely. Anything worth doing is going to require hard work.

But the truth of the matter is that you’ll most likely have to train them. I hope that this book will help you in that process and hopefully more books will be written with similar goals.

Finding Jobs

One thing I’ve always prided myself on the fact that I always stay up to date with technology. Since, I can’t know all technologies, I have to pick and choose. The way I did this early in my career was to look at the industry and try to predict what skills would be in demand in the next 5 years.

So, will Functional Programming be a requirement in 5 years? If you asked me at the time of this writing, I’d have to say that it probably won’t be. Even though many programming languages have adopted Functional features, e.g. Java now has Lambdas, it’s still difficult to find jobs where you can program in a Functional Language.

Stories of the benefits of using Functional Programming Languages in Greenfield Projects, i.e. projects that are written from start, will need time to be told and retold before the industry will start to consider this approach. Also, more books to help developers get up to speed quickly will need to be written.

That’s why I believe that somewhere between 2020 and 2030, Functional Programming will become the paradigm of choice for most Greenfield Projects.

Even then, there will still be a huge need for Imperative Programmers. There’s still a need for Cobol

Programmers some 60 years after its incarnation, but these jobs will be maintaining antiquated legacy systems.

Having said all of this doesn't detract from the fact that Functional Programming Languages are readily available today with more than sufficient ecosystems and are being used in Production for Artificial Intelligence, Finance, Cryptocurrencies, Web Services, and Web Applications, to name a few, in companies like Facebook, IBM, Twitter, AT&T, Bank of America, Barclays Capital, NVIDIA, Microsoft and many smaller unknown companies all over the world.

Also, keep in mind that Functional Programming ideas have begun to permeate the language landscape. Many more languages are adopting these concepts. It's only a matter of time.

The programming world is becoming more functional every day. It's better to be ahead of that curve.

Who is This Book For?

This book is for the working programmer. However, that doesn't mean that students cannot benefit. They just won't appreciate the benefits like someone who has suffered and struggled with one or more Object Oriented Programming languages.

My idealized reader is someone who has used at least one of the following languages or something similar: Javascript [6], C#, Java, Python, Ruby, etc. They have been working for at least 2 years as a programmer. They value the same things I listed earlier and are looking to improve themselves and make programming a more enjoyable experience.

They want the most powerful tools at their disposal and are willing to put in the effort upfront to become skilled at using them.

This idealized reader is just the imaginary person that I've written this book for. If that's not you then it's more a sign of a lack of my imagination than yours. Please don't let that discourage you.

From A to Z

I've purchased thousands of dollars worth of books in my career and I've read many different types of books. Here how I'd characterize some of them:

A, B, X, Y, Z

My favorite complaint for many technical books is this:

- Chapter 1: Your A, B, C's
- Chapter 2: Your 1, 2, 3's
- Chapter 3: Differential Calculus

These books always start off simple and lull you into a false sense that you'll be able to follow every step along the way. The beginning is easy and understandable. So far so good. Then, inevitably, you turn a page and hit a wall.

You frantically turn back to the previous page wondering if you missed something. Maybe your eyes glazed over while you were reading and if you just reread the last few paragraphs, it'll be a smooth transition into the next section.

Alas, it is not to be. You have gone from ***A*** to ***B*** to ***X***, ***Y Z***.

A, B, C

Then there are books that are just not enough. They do a great job of explaining the basics, but don't contain enough information to make you proficient in the subject matter.

I felt this way with many Haskell books. I read them and thought that I had a pretty good understanding. I'd then go online and look at some Haskell code and be completely and utterly lost.

The reason there are so many books like this is because it's easy to talk about simple things and in the beginning of any subject is the easy stuff. So many books that you try to learn from will leave you hanging.

They start off at ***A*** but leave you coming up short at ***C***.

X, Y, Z

Then there are books that you know are wrong for you right from Chapter 1. You read the Preface and Intro and felt that everything was going to be normal. That is until you cracked open the first chapter.

These books start at ***X*** and are clearly advanced.

A to Z

It's easy to claim to take someone from ***A*** to ***Z*** but what's difficult is understanding what exactly is meant by ***Z***.

When I think of ***Z***, I think of someone who could be hired as a Functional Programmer in a Purely Functional Programming Language to write standard Business software, e.g. Front end Web GUIs, command line programs, servers (web or otherwise), etc.

Z should represent what most of us do in our everyday work but using a Functional Language instead. ***Z*** does NOT mean that there isn't more to learn. One thing about Programming is that there's always more to learn.

The goal of most people when they learn a new technology is to be able to use it to solve problems that they encounter daily. That is what ***Z*** means to me and this book will take you to that ***Z***.

My goal here is to take a proficient programmer, i.e. someone who can write code and has done so multiple times with varying levels of success, and take them baby step by baby step from the basics through intermediate levels at the same rate all the way through to advanced until they can regularly take real world problems and use Function Programming Languages to efficiently and effectively solve them.

Why PureScript?

There are 3 reactions that I imagine people will have when they find out this books is going to use PureScript:

1. PureScript? What is that?
2. PureScript? Why not Haskell or Elm or …?
3. PureScript? Why PureScript?

So let me answer them one at a time.

What's PureScript?

PureScript is a ***purely*** Functional Language that compiles to Javascript. [7]

It can run in both the Browser and on the Server (via `Node.js`). Pretty much anything that you can write in Javascript can be written in PureScript.

PureScript has stolen a lot from Haskell. In fact, the syntax coloring for Markdown and AsciiDoc (what this book's written with) simply use Haskell's. Most of the developers behind PureScript got their start in Haskell and the PureScript compiler is written in Haskell (as is Elm's compiler).

PureScript has also improved on Haskell making it more powerful in some cases but also making it easier to use.

If you already know Haskell, you can learn PureScript very quickly. If you learn PureScript you can easily learn Haskell and you've surpassed learning Elm.

To get started with PureScript takes very little setup and can be accomplished in as little as a few minutes.

Why not Haskell or Elm or …?

Haskell

Haskell is an old language at 30 years of age. It started off as a research language and has lots of language extensions that one must learn to do modern programming. This complexity can distract the student from learning the concepts of Functional Programming.

PureScript has no such extensions.

Haskell also doesn't fare well in the Browser. Sure there are technologies that allow Haskell code to run in the Browser but as anyone in the Haskell community will tell you, they are not for the faint of heart and definitely not for the student.

So if you're a Front End Developer, Haskell doesn't really offer much hope in your everyday work.

PureScript on the other hand, was built from the ground up with the Browser in mind.

Haskell's setup, while greatly improved over the years, is a bit more heavyweight than most other languages. There are so many different options for editor plugins and build systems that it becomes difficult to determine which to use and their reliability is often substandard or requires the student to compile it from source code.

PureScript has only a couple of ways and, in this author's opinion, one good way to build PureScript programs. The choice for editor plugins is smaller and more reliable and has pretty good support for modern day editors.

Haskell has 3 types of Strings (technically 6). This flexibility allows a Haskell developer to use the Type of String that best suits the situation but many times multiple Types of Strings are used in a single application. This complexity can be difficult and distracting for students.

PureScript only has one Type of String.

Elm

Elm is a great ***A***, ***B***, ***C*** language but it doesn't take you all the way. It's meant only for Front End Web work so anyone working on the Backend will not be able to transfer what they learn to their everyday work.

Elm is very opinionated and dictates a single architecture to all Elm Programs. This architecture was developed with Web Front Ends in mind. In my opinion, it is not a great architecture for Front Ends and it's worse for Backends.

Some Other Functional Language

I could give you lots of reasons for why not some other language. Like why I think Scala is a compromise language that prioritized interoperability with Java and because of that, is unnecessarily verbose and complex. Or like how F# only works in Microsoft's .NET virtual machine and like Scala is not a Pure Functional Language.

I could make equally disparaging remarks regarding ReasonML, Exlir, Clojure, Idris, Agda, etc. but the truth of the matter is that I only know Haskell, PureScript and Elm.

To be fair, part of the reason for only knowing these languages is because of the reasons that I just mentioned. I've looked at the others and, in my opinion, they all fall short in one area or another so I didn't learn them.

At the moment, I only program in Haskell for the backend and Elm or PureScript for the front end at work, which gives me a lot of experience with using these languages to solve real world problems.

In the past, I've also programmed in Assembly, C, C++, Pascal, FORTRAN, Cobol, Smalltalk, Java, and Javascript to name a few. I'm am by no means a language expert, but I've used my share of languages to solve real world problems.

Now that I've probably disparaged a few reader's favorite environments (JVM, .NET) or languages, I want to say that the concepts you'll learn here are completely transferable to most, if not all, of the aforementioned.

For example, the Monad shows up in Scala, F#, Elixir (in the form of Libraries) and even in Javascript (Promise is a Monad). So learning these concepts will directly translate over into every one of these languages I so quickly dismissed.

Learning useful ideas is never a waste of time even if you can't use them right away.

Why PureScript?

I've sort of explained along the way the benefits of PureScript related to Haskell or Elm but there are 2 major reasons I chose it over the other two.

First, it's complete. For everything that I want to teach in this book, only Haskell and PureScript are advanced enough that they would work. Keep in mind that both languages are more advanced than what you'll learn in this book. This is a good thing since it means that you can keep growing on your own.

Second, PureScript can run on both the front end and the backend. Haskell only runs on the backend (with much pain on the front) and Elm only runs on the front end (with even more pain on the backend).

Four Part Harmony

This book is broken into 4 parts:

- Beginner
- Intermediate
- Advanced
- Beyond

Beginner

This section will slowly introduce the reader to Functional Concepts that are general to the paradigm and found in nearly all Functional Programming languages while introducing simple illustrative examples in PureScript along the way.

If you already have some Functional experience, you may be tempted to skip this section but I'd suggest that you do not. For two reasons.

First, this is where I will introduce you to PureScript and you'll get a chance to do some coding in PureScript.

Second, and most importantly, reviewing what you already know is always a worthwhile venture. It helps strengthen what you know and many times when I've done this, I've seen something that I thought I fully understood in a different light, solidifying my understanding even further.

Intermediate

Armed with a Functional Foundation and some experience writing simple functions in PureScript, this section will expand on that by delving into Typeclasses, Folds and other Mathematical Concepts that show up in Functional Languages, e.g. Functors.

Don’t worry if you’re Math phobic, any and all Math concepts will be fully explained with the assumption that the reader doesn’t have the requisite background.

Category Theory is a very small portion of this book. I almost left it out completely but if you hope to exist in this field, you will definitely bump into people who do understand Category Theory as it applies to Programming and it will do you good to have a general idea of what it is and how it applies to Functional Programming.

Advanced

I joke that when I learned C, the first thing I learned was "Hello, world!" and when I learned Haskell, it was the last thing I learned.

There’s a lot of truth in this. Doing I/O in a Purely Functional Programming language is actually Advanced. Seems crazy at first but by the time you get to that part of the book, I hope that it won’t seem so.

This section will delve into the everyday kinds of abstractions that are used to get real work done. The kinds of things you’ll encounter on a daily basis. It will give you the tools to accomplish these tasks effortlessly and you’ll wonder how you ever got along without them.

Beyond

In this section, we’ll build a backend server and its corresponding front end. This will be a very simple program but will help you see how to use full-stack PureScript. We’ll use the ***Halogen*** framework for the web interface on the front end and the ***HTTPure*** framework to build our web server on the backend.

Our front end will run inside the Browser and our backend will run inside of Node. [8]

Exercises

I hate Exercises in books. It all started with my Math books in school. They’d only give you the answers not how they solved the problems or what they were thinking along the way or what was learned by solving them. All we got were answers.

To make matters worse, most books would only give you the answers to the odd numbered questions most likely, to discourage cheating, which makes no sense when nearly every Math teacher on the planet lives by the Mantra "Show Your Work".

So the Exercises presented here will be different. I will first give you the problem. Then you will stop reading and try them on your own. If you get stuck along the way, return to the book and keep reading.

Sometimes the very next section will provide a hint but there will always be a section on how to code the implementation one line at a time. While we code together, I will explain the thinking behind each and every step. And when there are reasonable alternative solutions, I will give you those as well.

I will do this for every part of the book, not just a few exercises in the beginning.

So please, please, if you're like me and usually skip the Exercises, I implore you to work through them. I cannot tell you how many times I read something or listened to a lecture and thought I understood everything only to find my self frozen the moment my hands touched a keyboard.

Everything seemed to make sense when I was reading or watching the learning materials but the fact that I couldn't use this new found knowledge told me that I really didn't understand it.

My goal with these Exercises is to take what you've just learned and use it immediately to help solidify it in your mind and make it your own. Along the way, we'll discuss and think about what we just did and what kind of thinking goes into being able to write such code. Sometimes we'll contemplate larger concepts from only a few lines of code in order to really hammer home the right abstractions and models by doing a post-mortem analysis of our work.

Most books don't help you go from a blank page to a complete solution. This book will not be one of those. You will stare at many blank pages and when you get stuck, this book will take you one step at a time, i.e. one line of code at a time, from nothing to a working solution. That's one of the reasons this book is so large.

I've also put in a lot of mistakes. Some I made when I was coding. Others were added on purpose. This is why you will see hundreds and hundreds of compiler errors. Too many books give you an antiseptic experience where all errors have been removed making the process of programming seem like magic or a skill that only a select few can truly master.

I don't care how long you've been programming. Everyone makes mistakes and the sooner that you get used to the compiler errors and determining how to fix them, the better.

So Many Pages

Because of how I've chosen to write this book, it comes in at a hefty size. Please do not be discouraged by this. There are many reasons for the large size.

For one, when I show you how to code the exercises, I will write one line at a time and sometimes only a fraction of that line. Each time I do, I will show you what our function looks like at that step so most of the code gets copied over and over again.

I will also copy code from earlier to minimize the need for scrolling back through the book to understand what is being referenced. This helps you while reading, but definitely adds pages.

Another thing I do is break up the paragraphs into very few sentences. This helps give the reader time to digest the previous idea before being thrust into the next. I'll do this even if the paragraph only has a single sentence.

And finally, the book is mostly code and I've chosen a large font to reduce eye fatigue.

The goal of the look of this book is to have a lot of whitespace. This is to help reduce the amount of effort it takes to read it. The most important part isn't to keep the page count down but to make your job learning the subject matter as easy as I possibly can.

What you bring to the table is far more important. The effort you put in will be directly proportional to the value you get out. Remember to give yourself time; don't rush through the material.

All non-trivial subjects and skills take time. So be patient with yourself, work diligently and take breaks. A lot of learning happens in our sleep as our brains replay the day's events.

Do this and I promise that you will not regret it.

[1] Languages like Javascript, C#, Python and Java are Imperative Programming Languages.

[2] Here ***Z*** represents good enough to write in a Functional Language professionally. My personal goal was to write a Web Server in Haskell.

[3] See Why Functional Programming Matters by John Hughes for one approach to answering this question.

[4] See Phillip Wadler's talk Propositions as Types on YouTube or read his paper of the same name.

[5] We do still have Runtime Errors in our application, but it's almost always a crash statement that we put in to signal that our program logic is flawed or it's in our Javascript code since not everything we need to do can be written in Elm.

[6] Many early examples assume a modicum of Javascript knowledge.

[7] At the time of this writing there are other actively maintained backends to the compiler, e.g. C++11, Go and Erlang.

[8] At the time of this writing, Deno is still in its infancy but I suspect PureScript bindings will emerge for it some point in the near future.

Part I: Beginner

Chapter 1. Discipline is Freedom

In the mid-to-late 1960s, the Structured Programming paradigm aimed to improve the quality of programming while reducing the development time needed. One of the most notable features of Structured Programming Languages such as ALGOL, Pascal and C was not a feature at all. It was the absence of the GOTO statement.

This was highly controversial and hotly debated through the 1970s and well into the 1980s. The arguments against the elimination of GOTO usually focused on problems of the past, e.g. the number of machine instructions produced by GOTO-less code was more than code with GOTOs. The biggest opponents of this movement were, not surprisingly, experienced programmers who were used to using GOTOs to short-circuit their logic whenever it was most convenient. There was less programming overhead to simply pull the trigger on a GOTO and they were not willing to give that up for something they saw as less powerful.

In the examples routinely put forth by proponents, GOTO would prove to be more powerful and more terse than their structured counterparts. However, in practice, this was not the case. The term **Spaghetti Code** described the overuse of GOTO's, which made following a single control flow in a program as difficult as following a single strand of pasta in a bowl of spaghetti.

Today, no one debates the benefits of not using GOTO and it's been decades since a language came out with such a powerful yet dangerous feature. I can personally attest to the fact that we are far better off with its demise. I remember trying to follow Assembly Code, which, by its low-level nature, can only branch with a form of GOTO via Jump Instructions. If the programmer overused Jump Instructions, then the program was an incomprehensible mess and rewriting was much easier than trying to understand how it worked.

You may wonder how programs were written at all in such a wild-west sort of mentality. In the case of Assembly Language, the programmer just had to be very disciplined about control flow. In the case of higher-leveled languages, one would choose a language that was absent of such destructive options. With such options removed, you couldn't accidentally (or purposely) do something that you'd later regret.

This process of using discipline in the absence of appropriate built-in safety measures followed by language developers adding such features into new languages and retrofitting old ones, has been going on ever since.

It's at this point, you may be wondering, what things have I gotten used to doing in my programs that we'll all look back on as dangerous and unnecessary.

1.1. Global State

Global State (or Global Variables) has been known to cause problems in programs for quite some time. Some of these problems are:

- Anyone can change the data at any time in any module
- There can be tight coupling between Global Variables that can easily be broken by non-compliant code
- Concurrency requires contention-management logic that's not enforced

- Variable name collisions can occur since the Variable names are in every namespace

Many would argue that Object-Oriented Programming solves these problems. Unfortunately, it does not. It only hides them.

You can easily create a Singleton Object that contains all of your Globals and then expose them to be public. Now you have the equivalent of Global Variables in any Object-Oriented language. They are simply in the namespace of that Singleton Object, which itself is a Global.

Like with GOTO, the only real way to remove this scourge is to make it impossible to have Global State and Functional Programming Languages do exactly this.

1.2. Mutable State

The evils of Mutable State have also been known for quite sometime and recently we've seen many popular frameworks and languages moving, albeit slowly, towards giving developers tools to help control this.

Some of the problems caused by Mutable State are:

- Code is much harder to reason about since values can change drastically altering the code Semantics
- Code is more fragile since program reasoning is greatly reduced

To partially address these issues, Javascript added `const` (with all of its flaws) and libraries have been developed to support Immutable Data Structures, e.g. the `immutable.js` library.

However, these language features require the developer to be disciplined on a daily basis to avoid mutations and unfortunately, many of the standard library functions will mutate your data structures whether you want them to or not. It's also really easy to use the wrong Function, e.g. `splice` mutates but `slice` does not. In Javascript, you're just one typo away from having a mutation bug in production.

The popular framework React has a general immutable approach to its architecture. The problem is that it does so in a language that is anything but, putting the burden on the developer.

In Functional Programming, everything is immutable but what are the ramifications of removing mutability?

1.2.1. No Variables

Do you remember when you first learned programming and someone showed you the following:

```
x = x + 1
```

They told you to "Forget everything you learned in Math". While it's true that in Math, there is no solution for this formula, i.e. there is no value of `x` such that it equals `x + 1`, in programming, this means something completely different. It basically says, take the value of `x`, add `1` to it, and then place it back into `x`.

In these kinds of programming languages, `x` represents a placeholder in memory where numbers can be stored. We look to the right side of the equation first and evaluate it much like we would in Math and then we take the result of that evaluation and place it into a memory location named `x`.

In Functional Programming, there are no Variables. We still call them Variables like we did in Math but they don't vary like we are used to in Programming. Our thinking has to go back to middle school where we learned Algebra.

Remembering our Algebra word problems:

```
Mrs. Johnson has 20 students in her class. 12 of the students have brown hair,
3 have red and the rest have blonde. How many students have blonde hair?
```

Our fist step in solving this is to define our "Variables". So we'd write down the following:

```
Let s = number of students
Let b = students with brown hair
Let r = students with red hair
Let y = students with blonde hair
```

Then we'd write our formula:

```
y = s - b - r
```

To do this in PureScript, we write something very similar:

```
let s = 20          -- number of students
    b = 12          -- students with brown hair
    r = 3           -- students with red hair
    y = s - b - r -- students with blonde hair
```

Just like in Math, the variable `s` stands for `number of students`. The number of students doesn't change in the middle of our problem. It's always `20`. In Math and in Functional Programming `s` is immutable.

So how did we solve for `y` in Math?

We substituted:

```
y = s  - b  - r
  = 20 - 12 - 3
```

In Functional Programming, the compiler does the same substitution that we learned as kids and we can do this at any time since we don't have to worry that `s` or `b` or `r` will change.

When we say `s = 20`, we aren't asking the compiler to allocate some memory for our `20` and then call it `s`. What we're doing is defining `s` as the value `20`. Technically, `s = 20` is an ***expression***.

An expression is ***Referentially Transparent*** if it can be replaced with its value without changing the program's behavior.

Here `s` is defined to be the same as `20`. It's equal to `20` and since they are equivalent, we can use them interchangeably and whenever we see fit. In Haskell, `s` will be replaced only when it's needed since the language is by default ***lazy***, i.e. it only evaluates what it needs when it needs it.

PureScript by default is ***strict*** meaning it eagerly evaluates all definitions, which means we don't automatically benefit from this runtime optimization. However, we still get all the other benefits of Referential Transparency.

By removing mutability from our language, we gain the ability to do simple substitution at any time we see fit. We also can reason more easily about our code and in many cases can understand code better by doing simple mathematical substitutions that we learned in school.

It's at this point you're probably asking yourself, "How do I do a loop in a language with no Variables". The answer is you can't.

1.2.2. No Loops

In Math, there are no loops but it's still a powerful enough language (yes Math is a language) to define things like loops found in programming.

Let's look at the Factorial Function in Math where `n > 0`:

```
n! = 1 · 2 · 3 ··· (n − 2) · (n − 1) · n        (Eq. #1)
```

So then 5! is:

```
5! = 1 · 2 · 3 · 4 · 5
```

When we write this backwards:

```
5! = 5 · 4 · 3 · 2 · 1
```

we can see the following grouping:

```
5! = 5 · (4 · 3 · 2 · 1)
```

where the parenthesized value is just `4!`:

```
5! = 5 · 4!
```

which leads us to an alternative definition for Factorial:

```
n! = n · (n − 1)!
```

But what happens when `n − 1` is `0`?

How do we calculate `0!`?

We can figure this out by looking at `1!`:

```
1! = 1 · (1 − 1)!
   = 1 · 0!
```

From our original definition of Factorial (see `Eq. #1`), `1! = 1`. So, let's substitute and reduce:

```
1! = 1 · 0!
1  = 1 · 0!
1  = 0!
```

Therefore, we can reasonably define `0!` as:

```
0! = 1
```

So here's our Mathematical Definition of Factorial:

```
0! = 1
For all n > 0, n! = n · (n − 1)!
```

In more Mathematical Symbology:

```
0! = 1
∀n ∈ ℕ, n! = n · (n - 1)!
```

where `∀n ∈ ℕ` means `For all values of n that are elements of the set of Natural Numbers`.

Natural Numbers are also known as Counting Numbers, `1, 2, 3, 4, …`, which is equivalent to `n > 0` from our previous definition.

So how would we define a Factorial Function using loops (in Javascript):

```
const factorial = n => {
  var result = 1;
  for (var i = 1; i <= n; ++i)
    result = result * i;
  return result;
};
```

First notice how complex this definition is compared to our Mathematical one. You might dismiss this as being the expected difference between Math and Programming but it doesn't have to be that way.

You may expect that Functional Programming will magically reduce this complexity. Well, yes and no.

It's not a special feature of Functional Programming that reduces the complexity. Instead, it's just a different way to think about the problem that even most Imperative Programming languages support.

Imperative Programming languages, like Javascript, consist of a set of commands that change the program's state.

Looking back on our final Mathematical definition of the Factorial Function:

```
0! = 1
∀n ∈ ℕ, n! = n · (n - 1)!
```

We can see that Factorial is defined in terms of itself, i.e. there's a Factorial on the right side of the equation. This is known as a ***Recursive Definition***.

In programming, we can write recursive functions (here in Javascript):

```
const factorial = n => n === 0 ? 1 : n * factorial (n - 1);
```

With recursion, our Function parallels the Mathematical definition and is as easy to understand as its

Mathematical counterpart.

Turns out that you don’t need loops if you have recursion in your language and all Functional Programming languages support recursion since they don’t have loops.

Also, you can turn any loop into a Recursive Function and any Recursive Function into a loop.

The upside to Recursion is that many times it simplifies the Function making it easier to think about than a loop.

A downside to Recursion is that it’s not as intuitive as loops but with practice it gets a lot easier. We’ll do a whole set of exercises writing recursive functions. So much so that it’ll become second nature to think recursively.

Also, many recursive functions have the same pattern so there are library functions that help us leverage these similarities. We will examine these functions in detail as well.

1.3. Purity

Let’s look at some simple Mathematical functions:

```
f(x) = 25x − 17
g(y,z) = 42y + z
```

They do the following:

- Take one or more inputs
- Perform a computation
- Return one result

```
f(0) = −17
g(1,2) = 44
```

Functions in Math are very rigorous. They all abide by these restrictions.

However, in Imperative Programming, functions have no such requirements. We’ve all written functions that return no values. For example, in Javascript:

```
var workCounter = 0;
const doWork = str => {
  ++workCounter;
  console.log("Working with: " ++ str);
};
```

Notice that there is no `return` in this Function. Its return value is `undefined`. So why call it?

We call it for its Side Effects, which are:

- Increment `workCounter`
- Write to the `console`

In Purely Functional Programming Languages based on Lambda Calculus, we have the following rigorous requirements :

- Take one parameter
- Perform a computation
- Return one result

All functions like this are called **Pure**.

A ***Pure*** Function has NO Side Effects and given the same input will ALWAYS produce the same output.

Notice that these requirements differ from the Mathematical ones. Here our functions can only take a single parameter. You might think that this is terribly limiting but it's not.

Turns out that any Function with multiple Parameters can be rewritten as a set of functions that each only take a single parameter. This is a powerful concept called ***Currying***, which we will delve into more later.

Once again, Functional Programming has taken away something that we've all gotten use to, doing Side Effects whenever and wherever we like. If you think about it, any program that does NOT produce Side Effects is producing no valuable work.

It just runs for a while heating up the CPU and then exits with a single value. No files are written. No HTML code is sent or received. No updates are made to the Database. So how is it possible to write anything of any value if you're restricted to only Pure functions?

Trust me, you can. My first real Haskell program was a backend server that had a WebSocket API and maintained a Postgres Database. You'll have to wait a bit to see just how since doing Side Effects is unexpectedly a more advanced subject.

I often joke that when I learned C, the first thing I learned was Hello World and when I learned Haskell, it was the last.

In the early days of Haskell, there was no way to do Side Effects until they figured out how to do them in a Pure way. I know this sounds like an oxymoron but it's not as you'll see later.

1.4. Optimization

When you only use Pure functions, there are some hidden benefits that you get, viz. the ability to use certain optimizations.

Bad designs produce unexpected consequences whereas good designs produce unexpected benefits.

1.4.1. Memoization

This hard-to-pronounce term defines a technique of storing the results of expensive calculations so that the result can be looked up instead of recalculated.

To better understand this, let's look at a toy example in Javascript:

```
const g = v => {
  for (var i = 0; i < 10000000; ++i) {
    // do some complex calculation ❶
    console.log ("processing index: " + i); ❷
  }
  return v;
};
```

❶ Imagine some really long complex calculation here.

❷ Thanks to this line, `g` is not a Pure Function.

Unfortunately, since `g` is not Pure, we cannot Memoize the computation. But why?

If we call `g` twice with the same `v`, it will log to the console twice.

If we call `g` once with `v` and remember its result (memoize) and then if we simply look up its stored result instead of calling it a second time, we will only log to the console ONCE.

Our program will not behave the same and so we cannot optimize by using Memoization. One could argue that this difference is trivial but just imagine that our Side-Effect was writing to a Database.

Let's change `g` to be a Pure Function and build a Memoized version of `g` called `mg`:

```
const gCache = {}; ❶
const mg = v => {
  if (gCache[v] === undefined) { ❷
    const result = g(v); ❸
    gCache[v] = result; ❹
  }
  return gCache[v]; ❺
};
const g = v => { ❻
  for (var i = 0; i < 10000000; ++i) {
    // do some complex calculation ❼
  }
  return v;
};
```

❶ This is our cache of previously calculated values indexed by our input parameter (I know that this is global but this is Javascript and just an example).

❷ We check to see if we've already called `g` with the parameter `v`.

❸ If we haven't called `g` before, then we do so now and save the `result`.

❹ We put the `result` into our cache.

❺ We return the cached value.

❻ `g` is now Pure since we removed the Side-Effect of logging.

❼ Let's assume this complex calculation is also Pure.

In this case, we don't call `g` directly, unless we want to waste time computing, but instead call `mg`, our Memoized `g` Function.

The first time we call `mg` with `v`, the resulting computation won't be in our cache. So we compute it and save it indexed by the input parameter, `v`. If we call it a second time with the same input parameter, then we can simply look up the previous result and return immediately with our computation.

With Pure functions, an optimization solution could be hand-built or done by a smart Compiler via some form of annotation to mark `g` as expensive.

Without Purity, certain Compiler Optimizations are impossible.

1.4.2. Compiler Optimization

Let's consider another toy example in Javascript:

```
const g = v => {
  // do some complex calculation ❶
  console.log ("g was called with: " + v); ❷
  return v;
};

const f = n => n * g(x) - g(x); ❸
```

❶ Imagine we do a complex calculation here.

❷ `g` is NOT Pure due to this Side-Effect.

❸ There is a case for optimization here when `n = 1`.

It would be nice if the compiler would automatically optimize out the 2 calls to `g` when `n = 1` but it cannot since `g` is NOT Pure.

If `g` isn't called twice in `f`, then our program won't execute the Side-Effect of logging to the console twice. Even though we know the answer is always going to be `0`, we still have to do an expensive calculation twice.

Let's make `g` Pure:

```
const g = v => { ❶
  // do some complex calculation ❷
  return v;
};

const f = n => n * g(x) - g(x); ❸
```

❶ `g` is now Pure having removed the console logging.

❷ Same expensive computation.

❸ Now this can be automatically optimized.

The compiler can optimize `f` by generating code as if we had written the following:

```
const f = n => n === 1 ? 0 : n * g(x) - g(x);
```

Without Pure functions, this sort of automatic optimization is impossible since it's too difficult for the compiler to determine if optimizing will affect the program's behavior.

There are many more optimizations that compilers can make if it knows up front that a Function is Pure. These are just some simple examples.

1.5. Types

There are two kinds of Types in programming languages, ***Static Types*** and ***Dynamic Types***. Static Types are checked at compile time and Dynamic Types are checked at runtime.

Programmers have been debating which is better since the beginning of Programming. And it's no different today.

1.5.1. Signal to Noise

If you've worked with languages that have Static Types, e.g. Java, you may have gotten used to complex types such as:

```
private static final Map<Integer, List<String>> map =
  new HashMap<Integer, List<String>>;
```

The signal to noise ratio is too low. Scala, a Functional/OO hybrid, has equally noisy type definitions:

```
class NonEmpty[A](elem: A, left: Set[A], right: Set[A]) extends Set[A] {
  // redacted
}
```

So you can understand why programmers are willing to abandon Static Types for the simplicity of the following Javascript:

```
var map = {}
```

Or even simpler in Python:

```
dict = {}
```

However, there is a high price to pay for Dynamic Types:

- Type checking is deferred until runtime
- Tests to catch Type errors must be written and maintained, i.e. Technical Debt
- Refactoring is so risky that it's often avoided
- Nearly zero help from the IDE

One could argue that the downsides of Static Types are:

- Limits the programmer’s freedom
- Requires explicit Typing
- Signal to Noise is too low

1.5.2. Perceived Limits

There’s a common belief that with a Dynamically-Typed language, you don’t need to know what Type of data you will get until runtime and that this makes Dynamically-Typed languages more flexible than Statically Typed ones.

This just isn’t true. It’s impossible to write code that can deal with random data structures. What Statically-Typed languages force you to do is to be explicit with Types to guarantee that your program is correct.

Dynamically-Typed languages are great for a quick-and-dirty hack or for a very small program that can be debugged quickly and won’t be a maintenance headache, i.e. will never change.

Even so, if you care about code quality and correctness, then you’ll choose a Statically-Typed language because it will find a whole set of bugs at compile time as opposed to runtime. Pure Functional Programming languages that are Statically Typed can be automatically tested without burdening the developer with having to write, debug and support Unit Test code.

1.5.3. Cake and Eat It

Some languages force the developer to explicitly define every single Type but there are a whole set of languages, mostly Functional ones, that support ***Type Inference***.

Type Inference frees the programmer from explicitly defining Types since the Compiler can usually infer the Type based on its usage.

Take the following bit of PureScript code:

```
add x y = x + y
```

We can infer that x and y are some sort of Number. A Mathematician would say that x and y are members of a Semi-Ring (more on this much later).

So how did we "know" what Types were involved? We knew because of the + operator. It takes 2 Numbers and adds them together. We inferred based on how x and y were being used and this is, in essence, how the compiler infers types.

Specifying Types is important to help document code, especially with complex Types, but with Type Inference, we don’t always have to specify Types, e.g. in the case of short-lived, local Variables. This reduces the burden found in many Statically-Typed languages.

1.5.4. Cutting Through the Noise

Let’s revisit our Java example:

```
private static final Map<Integer, List<String>> map =
  new HashMap<Integer, List<String>>;
```

What makes this difficult to read is that the Type information is intertwined with the Variable names. In most ML-based languages, e.g. PureScript, Haskell, and Elm, they separate the Type information, i.e. the ***Type Signature***, from the Function definition.

Here’s an overly simplistic definition for `add` in PureScript:

```
add :: Int -> Int -> Int  -- Type definition
add x y = x + y           -- Function definition
```

Notice how the Types are defined on a separate line from the Function. This small change has a huge impact on our ability to understand what this Function does. In fact, sometimes we can tell everything about a Function by just looking at its Type Signature. We’ll spend some time looking at Type Signatures and guessing what those functions do later.

If you think that the Type Signature looks strange at first glance, you’re not alone. Don’t worry, we’ll spend plenty of time with Type Signatures until they become second nature.

1.5.5. Static Type Costs and Benefits

Languages that employ Static Types impose upon the programmer an extra burden to define Types and use them when creating new values. The benefit of this disciplined approach is the confidence gained that a program will operate as specified.

The code we will write in PureScript will be mostly debugged using the compiler. I’ve heard Haskell and Elm programmers say "If it compiles, it works." While this is an exaggeration, I can personally attest to this experience. More times that not, once my program compiles, it just works.

1.6. Summary

A disciplined approach to programming has proven itself over the years . It’s not surprising that such discipline is met with resistance from those in the profession who are under pressure daily to produce results. Eventually, the better ideas do make it to industry even if it takes a few decades.

Functional Programming’s restrictions are no different from the ones introduced by Structured Programming. Those restrictions made programming better back in the 1960’s as Functional Programming is doing today.

We’ve seen the restrictions that Functional Programming imposes on us in order to make our programs more stable and easier to understand:

- All functions are Pure (no Side Effects)
- All values are immutable (no Variables)
- Iteration is performed via Recursion (no Loop Constructs)

Next, we’ll take a look at some power features of Functional Programming that are seldom found in other paradigms.

Chapter 2. The Power of Functions

Way back when I was programming in Java, there were so many times I wanted to refactor my code, but could not. Here’s a simple example where I got stuck:

```
public class HelloWorld {
    public static String appendIfSmall(String s, String append) {
        if (s.length() < 10) ❶
            return s + append;
        else
            return s;
    }
    public static String appendIfOddLength(String s, String append) {
        if (s.length() % 2 != 0) ❷
            return s + append;
        else
            return s;
    }
    public static void main(String[] args) {
        System.out.println(appendIfSmall("Hello World", "!!!"));
        System.out.println(appendIfOddLength("Hello World", "!!!"));
    }
}
```

❶ This `if` checks to see if the length is "small".

❷ This `if` checks to see if the length is odd.

The output of the above code is:

```
Hello World ❶
Hello World!!!
```

❶ This doesn’t have `!!!` appended to it since it’s not "small" since its length is 11.

While this is a toy example, it demonstrates a very real problem. I cannot refactor this code into a single `append` Function. Here’s a failed attempt at doing so:

```
public class HelloWorld {

    public static String appendIf(String s, String append) {
        if (??? WHAT DO I PUT HERE ???) ❶
            return s + append;
        else
            return s;
    }
}
```

❶ I can't put BOTH the check for odd length and small length here. Besides, I don't want both checks. Sometimes I want the odd length check and other times I want the small length check. What I'd like is a way to pass in the `if` statement.

To solve this dilemma, we'll need something called a ***Predicate***, i.e. a Function that takes an input Parameter and returns a `Boolean`. Here are two Functions that encapsulate the two `if` checks.

```
public class HelloWorld {
    public static Boolean isSmall(String s) {
        return s.length() < 10;
    }
    public static Boolean isOddLength(String s) {
        return s.length() % 2 != 0;
    }
    // ...
}
```

What I want is a way to pass these Functions as a Parameter to a single `append` Function. Back when I was programming in Java, there was no way to do this. Now, Java has adopted a very powerful feature that Functional Programming has had for over 50 years, i.e. passing Functions as Parameters.

2.1. Functions as Parameters

In Functional Programming, Functions are first-class citizens. This idea has been around since the late 1950s. In Lisp, the saying goes:

Code is Data, Data is Code

This means that code can be manipulated like data and data can be executed.

While this is powerful, it's terribly dangerous and in very rare cases, e.g. Genetic Algorithms, it's a feature that one should generally avoid.

However, it's the idea that Functions are no different from `Integers` where the real power is. In our earlier `append` example, we struggled to pass a Predicate Function because the language simply had no mechanism.

Now that Java pilfered this feature from Functional Languages, we can write:

```
import java.util.function.Function;

public class HelloWorld {

    public static Function<String, Boolean> isSmall =
        s -> s.length() < 10; ❶
    public static Function<String, Boolean> isOddLength =
        s -> s.length() % 2 != 0; ❶
    public static Boolean isSmall(String s) {
        return s.length() < 10;
    }
    public static String appendIf(Function<String, Boolean> pred,
                                  String s, String append) {
        if (pred.apply(s)) ❷
            return s + append;
        else
            return s;
    }
    public static void main(String[] args) {
        System.out.println(appendIf(isSmall, "Hello World", "!!!"));
        System.out.println(appendIf(isOddLength, "Hello World", "!!!"));
    }
}
```

❶ These special types of `Functions` in Java can be passed to `appendIf` just like we passed the `Strings`, `s`, and `append`.

❷ We apply the Parameter, `s`, to our Predicate Function, `pred`.

I wish I had this facility back in the early days of Java but this kind of thing is commonplace in Functional Languages.

Let's look at the PureScript equivalent:

```
module Main where

import Prelude

import Data.String.CodePoints (length)
import Effect (Effect)
import Effect.Console (log)

isSmall :: String -> Boolean ❶
isSmall s = length s < 10

isOddLength :: String -> Boolean ❶
isOddLength s = length s `mod` 2 /= 0

appendIf :: (String -> Boolean) -> String -> String -> String
appendIf pred s append = if pred s then s <> append else s

main :: Effect Unit
main = do
  log $ appendIf isSmall "Hello World" "!!!"
  log $ appendIf isOddLength "Hello World" "!!!"
```

❶ These Functions are just run-of-the-mill Functions. No special syntax needed.

I don't expect you to understand this code just yet but I wanted to show you the terseness of PureScript compared with Java. Granted, if you're a Java programmer, the PureScript version will feel very foreign at first.

Regardless, there's a beautiful lack of parentheses, commas, semi-colons, curly-brackets and accessor types, i.e. `public` and `static`. While this may be jarring at this point, I promise that you will get spoiled by this very quickly.

Where Java uses `Function<String, Boolean>`, PureScript uses `String -> Boolean`.

In Java, there is a special protocol to use a Function that's been passed as a Parameter, `pred.apply(f)`. In PureScript, it's no different than calling any other Function, `pred f`.

Just for fun, let's look at the same code in Haskell:

```
module Main where

import Prelude hiding (pred) ❶ ❷

isSmall :: String -> Bool ❸
isSmall s = length s < 10

isOddLength :: String -> Bool
isOddLength s = length s `mod` 2 /= 0

appendIf :: (String -> Bool) -> String -> String -> String
appendIf pred s append = if pred s then s <> append else s ❷

main :: IO () ❹
main = do
  putStrLn $ appendIf isSmall "Hello World" "!!!" ❺
  putStrLn $ appendIf isOddLength "Hello World" "!!!"
```

❶ In PureScript, there's a finer granularity of libraries, so there's more to import.

❷ We hide `pred` from the import of `Prelude` because we don't want to conflict with our `pred` in `appendIf`.

❸ `Bool` is the Type for Booleans in Haskell.

❹ Instead of `Effect Unit`, we do effects in Haskell using `IO ()` where `Unit` is the equivalent to `()` in Haskell.

❺ In PureScript, we need to specially import `log` from `Effect.Console` because we could be writing for the Browser so `log` isn't as readily available as the Haskell equivalent, `putStrLn`.

While I've glossed over a lot of details, which we will get to later in this book, I hope seeing how similar PureScript and Haskell are will give you confidence that once you're done with this book, you can easily transition over to writing in Haskell.

Unfortunately, I cannot easily give you an example in Elm because Elm doesn't have facilities for outputting to the console since it can only run inside the Browser but the syntax is very similar.

The important thing here is that all three languages support the ability to pass Functions as Parameters.

2.2. Functions as Return Values

If Functions are Values and Functions return Values, can Functions return other Functions?

Yes we can and as it turns out, Javascript also supports Functions as Values. So here's an example in Javascript of a Function returning another Function:

```
const isEven = n => n % 2 === 0;
const upperLower = n => { ❶
  if (isEven(n)) return String.prototype.toUpperCase;
  else return String.prototype.toLowerCase;
};

console.log(upperLower(0).apply("this should be output in uppercase"));
console.log(upperLower(1).apply("THIS SHOULD BE OUTPUT IN LOWERCASE"));
```

❶ If n is even then we return a Function to convert strings to uppercase. Otherwise, we return a Function to convert strings to lowercase.

Let's see how we'd write this in PureScript:

```
module Main where

import Prelude

import Data.String.Common (toLower, toUpper)
import Effect (Effect)
import Effect.Console (log)

isEven :: Int -> Boolean
isEven n = n `mod` 2 == 0

upperLower :: Int -> (String -> String)
upperLower n = if isEven n then toUpper else toLower

main :: Effect Unit
main = do
  log $ upperLower 0 "this should be output in uppercase"
  log $ upperLower 1 "THIS SHOULD BE OUTPUT IN LOWERCASE"
```

Once again, the PureScript example will not completely make sense to you yet but we're only concerned with how it contrasts with Javascript.

It's also possible that you are not familiar with Javascript either and in that case, the best you can hope to discern from these two examples is that the syntax in PureScript is far less dense than in Javascript.

You may also notice that there's a lot more overhead in PureScript because we have to import libraries. In this particular example, Javascript has all of the Functions we need already built-in. This is not always the case.

If you've spent any time working with Javascript, you're well aware of how complicated adding libraries can be especially if you're targeting the Browser but using a library originally written for Node.

Such is the life of a front-end developer these days.

2.3. Higher-order Functions

To summarize, Functional Languages treat Functions as Values, which means that they can be passed to other Functions as Parameters and can be returned from other Functions.

The technical term for this is ***Higher-order Function***.

A ***Higher-order Function*** is a Function that takes a Function as a Parameter, returns a Function or both.

In contrast, Functions that do not do this are known as ***First-order Functions***.

Look back at the PureScript code and see if you can find the Higher-order Functions and the First-order Functions.

If you get stuck, try looking at the Javascript code. There are two First-order Functions and one Higher-order.

Let's see how you did:

```
module Main where

import Prelude

import Data.String.Common (toLower, toUpper)
import Effect (Effect)
import Effect.Console (log)

isEven :: Int -> Boolean ❶
isEven n = n `mod` 2 == 0

upperLower :: Int -> (String -> String) ❷
upperLower n = if isEven n then toUpper else toLower

main :: Effect Unit ❶
main = do
  log $ upperLower 0 "this should be output in uppercase"
  log $ upperLower 1 "THIS SHOULD BE OUTPUT IN LOWERCASE"
```

❶ First-order.

❷ Higher-order.

In PureScript, the syntax `(String -> String)` stands for a Function that takes a `String` and returns a `String`, and since this is the return type of `upperLower`, it means that `upperLower` is a Higher-order Function.

Don't worry if this isn't making much sense just yet, because we'll delve into Type Signatures later, which will clarify all of this. We're just dabbling in the code to help us get acclimated to PureScript.

2.4. Composition

Building larger things from smaller components is seen in nature, engineering, manufacturing and in programming. Building larger, more complex Functions by composing smaller, simpler ones is a very powerful technique.

In Math, we learned, and most likely forgot, that we can combine two Functions using Functional Composition:

```
f : Y → Z, g : X → Y ⇒ f ∘ g (x) = f(g(x)) : X → Z ∀x ∈ X
```

If you're not used to reading Mathematical statements, this can be daunting. So let's break this syntax down:

```
f : Y → Z
```

This states that `f` is a Function that maps elements from the Domain `Y` to the Codomain `Z`. Don't forget that `Y` and `Z` are Sets, e.g. `Y` could be the set of Real Numbers, `ℝ` (this is the symbol Mathematicians use to denote Real Numbers).

What Mathematicians call `Domain` and `Codomain`, we programmers call `Input` and `Output` respectively.

What does `maps` mean in this context?

It means that `f` takes an element of `Y`, traditionally depicted by `y` and produces an element of `Z`, usually depicted by `z`.

The arrow syntax is very similar to the arrow syntax we've seen used in Haskell and PureScript when defining its Functions. For example:

```
negate :: Int -> Int
negate n = (-1) * n
```

`negate` is just like `f` above. Its Domain or Input is `Int` and its Codomain or Output is `Int`.

Let's continue parsing:

```
g : X → Y
```

This states that `g` is a Function that maps elements from the Domain `X` to the Codomain `Y`.

Notice how `g's` Codomain is the same as `f's` Domain, i.e. the Output of `g` is the Input of `f`. This is a critical requirement if we are to *glue* these two Functions together.

```
f ∘ g (x) = f(g(x)) : X → Z ∀x ∈ X
```

The first portion states that `f` composed with `g` applied to `x` is equal to applying `x` to `g` and then applying those results to `f`. The `∘` symbol is the compose operator.

The remainder of this line states that the `f composed with g` maps elements from the Domain `X` to the Codomain `Z` for all elements `x` that are in `X`.

Functional Composition works from Right to Left.

```
f ∘ g
```

This is spoken as `f` ***COMPOSED WITH*** `g` or `f` ***AFTER*** `g` or `f` ***FOLLOWING*** g. The words ***AFTER*** and ***FOLLOWING*** help to remind us that composition is Right to Left.

```
f ∘ g (x)
```

This is spoken as `f` ***COMPOSED WITH*** `g` ***APPLIED TO*** `x`.

Now let's attack the whole thing:

```
f : Y → Z, g : X → Y ⇒ f ∘ g (x) = f(g(x)) : X → Z ∀x ∈ X
```

This states that if `f` maps `Y` to `Z` and `g` maps `X` to `Y` then `f` ***COMPOSED WITH*** `g` applied to `x` is equal to `x` applied to `g` applied to `f`, which maps `X` to `Z` for all `x` that are elements of `X`.

That's a mouthful when written out. You can see why Mathematicians use symbols like programmers do.

Speaking of programmers, how do we compose two small Functions into a larger one in code?

Here's an example in PureScript:

```
toString :: Int -> String ❶
toString n = show n

toArray :: String -> Array String ❷
toArray s = [s]

intToStringArray :: Int -> Array String ❸
intToStringArray n = toArray (toString n) ❹
```

❶ `toString` maps `Int` to `String`, i.e. give this Function an `Int` and it'll return a `String`.

❷ `toArray` maps `String` to `Array String`, i.e. given a `String` it'll return an array of `String`.

❸ Maps `Int` to `Array String`.

❹ `intToStringArray` is a MANUAL composition of `toString` and `toArray`. Notice how this takes an `Int` and uses `toString` to produce a `String` that is then passed to `toArray`, which produces an `Array String`, i.e. our return Value.

Let's rewrite `intToStringArray` using PureScripts Composition Operator, <<<:

```
intToStringArray :: Int -> Array String
intToStringArray = toArray <<< toString ❶
```

❶ Here we are composing the 2 Functions using <<<. Notice how the Composition Operator helps to remind us that Functional Composition works from Right to Left like it does in Math. But in PureScript, unlike Math, we can also go left to right using >>>.

Let's revisit a shortened version of the Mathematical definition from above:

```
f : Y → Z, g : X → Y ⇒ f ∘ g : X → Z
```

Notice how the ***OUTPUT*** of `g` matches the ***INPUT*** of `f`. The PureScript compiler would give us an error if `toString's` Output Type didn't match `toArray's` Input Type.

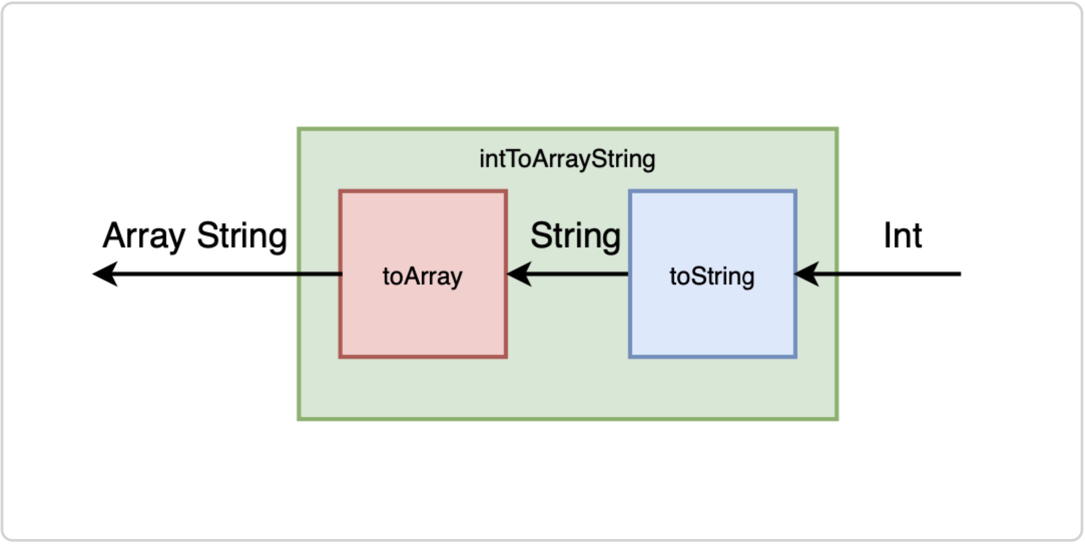
To help show how code fits together, take the following code:

```
toString :: Int -> String
toString n = show n

toArray :: String -> Array String
toArray s = [s]

intToStringArray:: Int -> Array String
intToStringArray = toArray <<< toString
```

and display it pictorially:



toString takes an Int and outputs a String, which is the input to toArray that outputs Array String.

Composing toArray <<< toString gives us intToStringArray which takes an Int and produces an Array String.

Composing Functions is a powerful feature of Functional Languages and is used often in most Functional codebases. Haskell uses the . operator to mimic the Mathematical ∘, whereas PureScript uses the <<< operator to help us remember that Composition in Math is from Right to Left:

```
notLarge :: Int -> Boolean
notLarge = not <<< isLarge ❶
```

❶ You don't need to use <<< with not in this example. It's just used here as an example. The code could have been just not isLarge.

PureScript also provides the >>> operator when we want to compose in the opposite direction. This is usually done because the code reads better. In the above example, it reads better using <<<. As we progress, we'll see examples where >>> is preferable.

2.4.1. Point-free Notation

Notice how our `notLarge` Function has NO formally defined Parameters. That's because it's written in a form called ***Point-free***.

Point-free notation (or style) means that a Function is defined WITHOUT explicitly mentioning one or more of its Parameters.

We could write `notLarge` by explicitly mentioning its Parameters but now we have the burden of naming and remembering them:

```
notLarge :: Int -> Boolean
notLarge num = not (isLarge num) -- NOT Point-free
```

This version is not as easy to read as our Point-free one. Writing a Point-free version of a Function doesn't always make the Function easier to read but when it does, it's definitely worth doing.

Most Imperative languages don't have direct support for Composition or Point-free Notation. For example, in Javascript, `notLarge` must be written like:

```
const notLarge = num => not (isLarge num);
```

Compare that with our PureScript, Point-free version:

```
notLarge :: Int -> Boolean
notLarge = not <<< isLarge
```

The Point-free version reads like English (`notLarge` is `not <<< isLarge`). This is why I chose to use `<<<` as opposed to `>>>`, which reads backwards (`isLarge >>> not`).

Now, let's assume that you wrote the following Function to pad a `String` with zeros on the Left to a specified size:

```
zeroPad :: Int -> String -> String
zeroPad size s = padLeft '0' size s
```

Is it possible to rewrite this Function to be Point-free? The fact that `size` and `s` appear on both sides of the equal sign should signal that it ***may*** be possible.

If you don't see how to write this Point-free, then maybe adding redundant Parentheses will help:

```
zeroPad :: Int -> String -> String
zeroPad size s = (padLeft '0') size s
```

If we look at our Function as a simple equation, we can see that `(padLeft '0')` is equal to `zeroPad`, giving us:

```
zeroPad :: Int -> String -> String
zeroPad = padLeft '0'
```

Now we have `zeroPad` in Point-free notation. It also reads better (`zeroPad` is `padLeft` with `0`).

Notice how we didn't touch the signature. That's because the Function still takes those Parameters. They're just Unnamed Parameters.

There's a simpler way to look at this sort of reduction using what I call ***Cancel on the Right***:

```
zeroPad :: Int -> String -> String
zeroPad size s = padLeft '0' size s
```

First, notice how `s` is the rightmost Parameter on both sides of the equal sign. That means we can cancel it from both sides of the equation:

```
zeroPad :: Int -> String -> String
zeroPad size = padLeft '0' size
```

Now `size` is the rightmost Parameter on both sides of the equal sign, so we cancel it as well:

```
zeroPad :: Int -> String -> String
zeroPad = padLeft '0'
```

Since there are no more Parameters on the left side of the equation, `zeroPad` is in its most reduced Point-free form.

The technical term for this reduction is ***Eta-Reduction*** (notated by `η−reduction`). The name comes from the use of the 7th letter of the Greek Alphabet, ***Eta***, which is uppercase `H` or lowercase `η`. This term is from ***Lambda Calculus***, which is an area of Mathematics that Functional Programming has "borrowed" heavily from.

One thing to keep in mind is that the order of Parameters matters for Eta-Reduction. Imagine that `zeroPad` had a slightly different definition:

```
zeroPad :: String -> Int -> String
zeroPad s size = padLeft '0' size s
```

In this case, we would be unable to reduce this because `size` is the rightmost Parameter on the left-side of the equation but `s` is the rightmost on the right-side. Just like in Algebra, we cannot cancel `size` on one side with `s` on the other.

Many times when coding, I'll change the order of my Parameters to allow for this cancelation to help reduce the complexity of my Function definitions.

2.5. Currying

Do you remember that one of the restrictions of Functional Programming languages is that all Functions only take 1 Parameter? Well, this restriction comes from Lambda Calculus, which was created by the mathematician ***Alonzo Church***. But the name comes from ***Haskell Curry***, who like Church set out to base Logic and Mathematics on Functions not Set Theory.

In Lambda Calculus, Functions can only take one Parameter. They are written in the following mathematical notation:

```
λx. x + 1
```

In PureScript, we'd write this as:

```
\x -> x + 1
```

Notice how the `\` sort of looks like a Lambda, `λ`. The `.` is replaced by `->`, but the idea is the same.

In both cases, this defines an anonymous Function that takes a single argument and adds 1 to it.

How do we write a Function in Lambda Calculus that adds two numbers if our Functions can only take one Parameter?

Don't forget that Functions can return other Functions:

```
λx. λy. x + y
```

Here is the PureScript equivalent:

```
\x -> \y -> x + y
```

When this Function is called with 3:

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

the value 3 binds to the variable x and returns the following Function:

```
\y -> 3 + y
```

When we call this returned Function with 5:

```
(\y -> 3 + y) 5
```

the value 5 binds to y returning:

```
3 + 5
```

since + is a Function, it returns:

```
8
```

In PureScript (and in Haskell and Elm) we can write:

```
\x y -> x + y
```

While this is illegal in Lambda Calculus, in programming, we are able to define multiple Parameters to reduce visual complexity. How is this possible when we stated earlier that all Functions in Functional Programming only take a single Parameter.

This is where ***Currying*** comes in. Currying takes a Function like:

```
\x y -> x + y
```

and turns it into a Function like:

```
\x -> \y -> x + y
```

The compiler does this for us ***automatically*** so we can have the nicety of writing Functions with far less cruft.

Let’s look at a named Function:

```
add :: Int -> Int -> Int ❶
add x y = x + y
```

❶ The Type Signature is Right-Associative, which implies that this Function is Curried.

All Type Signatures have implied Parentheses that are Right-Associative, i.e. associates to the right:

```
add :: Int -> (Int -> Int)
add x y = x + y
```

With the implied Parentheses explicitly defined, we can see how this Function only takes 1 Parameter and returns a Function that also only takes one Parameter, an `Int` and returns an `Int`.

Here’s another Function with more Parameters and explicit, Right-associative Parentheses:

```
add3 :: Int -> (Int -> (Int -> Int))
add3 x y z = x + y + z
```

With explicit Parentheses, we can see that Functions only take a single Parameter and return a Function that only takes 1 Parameter until the final Value is returned, which in the above case is an `Int`.

Here’s another example with explicit Parentheses:

```
show3 :: Int -> (Int -> (Int -> String))
show3 x y z = show x <> "," show y <> "," show z ❶ ❷
```

❶ `show` takes our `Ints` and turns them into `Strings`.

❷ `<>` concatenates our `Strings`.

When Parentheses are not right-justified, the meaning changes:

```
f :: Int -> (Int -> String) -> String
```

- The first Parameter is `Int`.
- The second Parameter is `Int -> String`
- The result is `String`

This means that the second Parameter is a Function, making `f` a Higher-order Function.

What about a Function like this:

```haskell
g :: ((Int -> Int) -> Int) -> Int
```

- The first Parameter is `((Int -> Int) -> Int)` which takes a Function from `Int -> Int` and returns an `Int`.
- The result is `Int`.

What are the implied Parentheses for:

```haskell
h :: Int -> String -> String -> String
```

Remember they are Right-Associative meaning associate using Parentheses to the right.

So you start on the right side and add the first set of Parentheses:

```haskell
h :: Int -> String -> (String -> String)
```

Then do it again:

```haskell
h :: Int -> (String -> (String -> String))
```

We can stop once we have a Function that takes only 1 Parameter. In this case, `h` takes an `Int` and returns a Function of type `String -> (String -> String)` so we can stop.

These implied Parentheses are there because Functions can only take a single Parameter. The important thing to note is that variables get bound along the way. For example, returning to `add3`:

```haskell
add3 :: Int -> Int -> Int -> Int
add3 x y z = x + y + z
```

When we call `add3` with one Parameter:

```haskell
add3 1 :: Int -> Int -> Int
```

`x` is bound to `1`. When we call that resulting Function with one Parameter:

```
(add3 1) 2 :: Int -> Int
```

y gets bound to 2. Calling that resulting Function with one Parameter:

```
((add3 1) 2) 3 :: Int
```

z gets bound to 3. The Function is now ***Fully Applied*** and returns a value of 6, which is of Type `Int`.

2.5.1. Partial Application

Returning to our `zeroPad` Function:

```
zeroPad :: Int -> String -> String
zeroPad = padLeft '0' ❶ ❷

padLeft :: Char -> Int -> String -> String
padLeft padCh size s = ... ❸
```

❶ `padLeft` is given only 1 of the many Parameters it will need to produce a final result.

❷ We have partially applied `padLeft`.

❸ We're not concerned with `padLeft's` implementation at the moment.

Partial Application is when a Function is called with fewer than all of its Parameters.

When we think of Functions having multiple Parameters, then Partial Application makes sense. However, when we think of Curried Functions, there is no such thing as Partial Application since there's only one Parameter.

The power of Partial Application may not be immediately apparent but consider a simple example:

```
pad :: Boolean -> Char -> Int -> String -> String
pad rightSide padCh size s = ... ❶

padLeft :: Char -> Int -> String -> String
padLeft = pad false ❷

padRight :: Char -> Int -> String -> String
padRight = pad true ❷

zeroPad :: Int -> String -> String
zeroPad = padLeft '0' ❸

spacePad :: Int -> String -> String
spacePad = padRight ' ' ❹

dotPad :: Int -> String -> String
dotPad = padRight '.' ❹
```

❶ We're not concerned with `pad's` implementation at the moment.

❷ Partial Application of `pad`.

❸ Partial Application of `padLeft`.

❹ Partial Application of `padRight`.

I like to think of Partial Application as configuring a more general Function. We're sort of ***baking*** in the configuration Values.

Notice that Partial Application is only possible because those "configuration Parameters" are specified first. `pad's` `Boolean` Parameter lets us define `padLeft` and `padRight` and then their `Char` Parameter lets us define `zeroPad`, `spacePad` and `dotPad`.

Here order is important. Imagine if the `Char` came before the `Boolean`:

```
pad :: Char -> Boolean -> Int -> String -> String
pad padCh rightSide size s = ...

padZero :: Boolean -> Int -> String -> String ❶
padZero = pad '0'

padSpace :: Boolean -> Int -> String -> String ❶
padSpace = pad ' '

zeroPad :: Int -> String -> String ❷
zeroPad = padZero false

spacePad :: Int -> String -> String ❷
spacePad = padSpace true

padDot :: Boolean -> Int -> String -> String ❶ ❸
padDot = pad '.'

dotPad :: Int -> String -> String ❷ ❸
dotPad = padDot true
```

❶ We have to write these 3 Functions to bake in the pad character first.

❷ Next, we write these 3 Functions that specifies which side to pad on.

❸ To implement `dotPad`, we have to write an extra Function, `padDot` since the pad character is `pad's` first Parameter.

This approach forced us to write a new Function for each new pad character. Our original approach was better since it let us create `padLeft` and `padRight` BEFORE adding the pad character. Now we can leverage those Functions anytime we want to add a new pad character Function.

It's worth spending some time planning the order of your Parameters for this very reason.

The general rule for Parametric Order is to have the Parameters that change the ***least*** be the ***leftmost*** ones and the ones that change the ***most*** to be ***rightmost***.

Revisiting the better version of our padding Functions:

```
pad :: Boolean -> Char -> Int -> String -> String
pad rightSide padCh size s = ...

padLeft :: Char -> Int -> String -> String
padLeft = pad false

padRight :: Char -> Int -> String -> String
padRight = pad true

zeroPad :: Int -> String -> String
zeroPad = padLeft '0'

spacePad :: Int -> String -> String
spacePad = padRight ' '

dotPad :: Int -> String -> String
dotPad = padRight '.'
```

If you look at `pad`, you'll notice that the `size` Parameter comes before `s`, the `String` we want to pad. This is because we may want to write the following Function:

```
dotPad20 :: String -> String
dotPad20 = dotPad 20
```

Now, we can pass a bunch of `Strings` to `dotPad20`. The `String` we want to pad is the thing that will change the most and, therefore, should be dead last in our Parameter List.

All of the other Parameters can be thought of as Configuration Parameters to tell `pad` how to pad whereas `s` is what to pad and since it will change the most, it's the rightmost Parameter.

Sometimes you'll have 2 Parameters that always change and, in that case, just make sure that they are last. Their respective order won't matter.

Chapter 3. The Basics of PureScript

Programming in C by Kernighan and Ritchie started a tradition when learning a new languge by starting with a program that prints `Hello, World!`.`

So here it is:

```
module Main where ❶

import Prelude ❷

import Effect (Effect) ❸
import Effect.Console (log)

main :: Effect Unit ❹
main = log "Hello, world!" ❺
```

❶ Every `module` starts with its definition. Here we called it `Main` and it would be placed in a file `Main.purs`.

❷ You import libraries using `import`. Here we're importing the standard base library called `Prelude`.

❸ We also import a few more libraries to do Side-Effects, i.e. print to the console. We will revisit Side-Effects in detail much later in this book.

❹ This states that `main` is a Side-Effect Function whose computation is `Unit`, the Functional equivalent of `void` in other languages, e.g. C, C++ and Java.

❺ This logs to the console our message to the world.

I/O like this is an advanced subject in Functional Programming, since it's a Side-Effect, there's a lot here that you won't fully understand just yet.

This shouldn't stop us from using the `log` Function for quick and dirty testing of the Functions that we write for exercises. I promise that once we get through Monads, this code will make a lot more sense.

To start with Monads would be like starting with High School before Kindergarten, so you'll just have to trust my superficial explanations for a bit.

Instead of digging into `Hello, world!`, we're going to start by learning about Types, one of the most important parts of PureScript and Haskell since they are both Statically Typed languages.

We'll return to the subtleties of `Hello, world!` in the Advanced part of this book.

3.1. Types

Most of PureScript's code is open source and can easily be perused via the GitHub repositories, but the implementation for Primitive Types are built into the compiler, which is written in Haskell.

In PureScript, Types always start with an uppercase letter.

3.1.1. Javascript Primitives

The following Primitive Types map directly to Javascript's types:

- Boolean
- Char
- String
- Number

Boolean Type

`Boolean` values are `true` and `false`. Unlike Haskell and Elm, these are true primitives since they compile directly to Javascript's boolean values.

```
t :: Boolean
t = true

f :: Boolean
f = false
```

Char Type

`Char` is defined with single quotes:

```
ch :: Char
ch = 'a'

unicodeCh :: Char
unicodeCh = '\x00E9'
```

String Type

`String` is defined with double quotes:

```
s :: String
s = "This is a multi-line string\nwith embedded newlines"

s2 :: String
s2 = "This is a multi-line string with continuations\
     \ at the end of the lines" ❶

s3 :: String
s3 = """ ❷
This is a multi-line that can contain quotes "" but \n will not be a newline
"""

unicodeStr :: String
unicodeStr = "This is a unicode character: \x00E9" ❸
```

❶ Characters to the left of `\`, in this case spaces, are NOT part of the string.

❷ Triple quotes can have newlines in them. The escape sequence `\n` will NOT be interpreted as a newline but will be interpreted as the literal characters, `'\'` and `'n'`.

❸ Unicode characters can be embedded with the escape sequence `\x` followed by a 4-hex-digit code.

Number Type

`Number` compile directly to Javascript representation for numbers:

```
import Prelude ❶

n :: Number
n = 1.0 ❷

smallestNumber :: Number
smallestNumber = (-5e-324) ❸

largestNumber :: Number
largestNumber = 1.7976931348623157e+308
```

❶ Imports `Prelude` for the negate operator, i.e. `-`.

❷ Must have decimal point otherwise interpreted as `Int`.

❸ Negative numbers MUST be in Parentheses. You will forget this. I know I still do from time to time. So, I'll say it again. Negative numbers MUST be in Parentheses.

3.1.2. PureScript Primitives

These are the PureScript-specific primitives, i.e. they don't map ***directly*** to Javascript's types:

- Int
- Array
- Record

Int Type

The underlying representation of `Int` is Javascript's number type but all of its operators, e.g. `+`, `-`, etc., will limit the results to only `Int` values:

```
import Prelude

i :: Int
i = 42 ❶

i2 :: Int
i2 = 1 + 4

smallestInt :: Int
smallestInt = (-2147483648) -- -2^31 ❷

largestInt :: Int
largestInt = 2147483647   -- 2^31 - 1
```

❶ No decimal point otherwise it'll be interpreted as `Number`.

❷ Just in case you've already forgotten, negative numbers MUST be in Parentheses.

Even though Javascript has a range of -2^{53} to $2^{53} - 1$ for Integers with no loss of accuracy, PureScript limits `Int` to 32-bits, i.e. -2^{31} to $2^{31} - 1$.

There is, however, a PureScript library called `purescript-int-53` that supports the full 53-bit Integer that Javascript supports. This increases the range to from `-9,007,199,254,740,991` to `9,007,199,254,740,991`.

Array Type

The `Array` Type is represented by Javascript arrays at runtime but, unlike Javascript, the elements must be ***Homogenous*** (***Homo***, meaning ***same*** and ***genos*** meaning ***a kind***), i.e. Arrays must contain elements of the same Type.

```
mta :: Array Number
mta = [] ❶

a :: Array Int
a = [1, 2, 3]

a2 :: Array String
a2 = ["abc", "123"]

aa :: Array (Array Int) ❷
aa = [ [1, 2, 3], [4, 5], [6, 7, 8, 9] ]
```

❶ An empty `Array` of `Number`.

❷ An `Array` of an `Array` of `Int`, i.e. array of arrays.

Record Type

`Record` is an aggregate Type with named fields that is represented by Javascript objects at runtime.

```
r :: { firstName :: String, lastName :: String }
r = { firstName: "Joe", lastName: "Mama" }

type Person = ❶
  { name :: String
  , age :: Int
  }

r2 :: Person ❶
r2 = { name: "Jane Doe", age: 37 }

type Nested = ❷
  { val :: Int
  , rec ::
      { val2 :: Int
      , name :: String
      }
  }
```

❶ You can define a Type Alias using the `type` keyword. Here `Person` is an alias for the Record. You can imagine copying and pasting the Record everywhere the Type Alias name is used.

❷ This is a nested record. The field `rec` is itself a Record. Notice the formatting of the records. They are defined on multiple lines. This formatting is idiomatic PureScript formatting.

Syntactical Oddity

To create a `Record`, we use the following syntax:

```
person = { name: "Candy Cane", age: 37 }
```

To modify a one or more elements in the `Record` and return a ***new*** `Record` (remember, all Values are immutable), we use the following syntax:

```
newPerson = person { name = "Randy Cane" }
```

This is the ONLY time an equal sign is used in `Record` syntax. All other times, a colon is used. This is another nuance that's easy to forget.

If you know Javascript, using a `:` won't be foreign but using the `=` for updating a `Record` will be.

3.1.3. User Types

PureScript comes with many useful built-in Types but without the ability to make your own Types, you won't be able to model your problem domain. PureScript has the following facilities for defining your own Types:

- Type Alias
- Data Type
- Algebraic Data Types
  - Product Type
  - Coproduct or Sum Type
- New Type

Type Alias

A Type Alias allows you to make an alias for another Type. Think of it as shorthand for a more complex Type.

```
type Id = String

type Message = { id :: Id, payload :: String }

type MessageHandler = Message -> Result

handler :: MessageHandler ❶
handler messageHandler = ...

handler' :: Message -> Result ❶ ❷
handler' messageHandler = ...
```

❶ These two Type Signatures are identical. The compiler substitutes `MessageHandler` with its definition, `Message -> Result`.

❷ PureScript allows you to name a Function with a ***prime***, i.e. the apostrophe character. Traditionally, this is at the end of the Function name and is related to the unprimed version in some way.

Data Type

We can make our own Data Types from scratch. Here is the simplest example:

```
data MyType = MyType
```

Notice `MyType` is on both sides of the equal sign.

The `MyType` on the left-hand side is defining a new ***Data Type*** called `MyType`.

On the right-hand side is `MyType` again but this defines the ***Data Constructor***.

The namespace for Data Types and Data Constructors are separate so there is no name collision here. You cannot use a Data Constructor where a Data Type is used and vice versa, which is why you will often see the names reused on both sides of the equal sign. Many times the word, Data, will be dropped and these will be referred to as Type and Constructor.

In PureScript, Data Types and Data Constructors always start with an uppercase letter, whereas variables start with a lowercase letter or an underscore:

```
x :: Int
x = 100

_b :: Boolean
_b = true

data MyOtherType = MyOtherType
```

Algebraic Data Types (ADTs)

There are 2 types of `Algebraic Data Types`, `Product Types` and `Coproduct Types` (also called `Sum Types`). The names reflect the process by which you determine how many inhabitants exist in a particular Type.

If the calculation involves a multiplication then it's a `Product Type`. If it involves addition then it's a `CoProduct Type` or `Sum Type`. In Math, the prefix `co` is added to mean the opposite (Domain, Codomain, Sine, Cosine, Tangent, Cotangent, etc.).

Sum Types (Coproduct)

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

In Haskell and Elm, this is how they define their boolean values. We can make our own `Bool` in PureScript since it won't collide with `Boolean`.

Here `Bool` is the Data Type and `True` and `False` are the 2 Data Constructors:

```
mkTrue :: Bool ❶
mkTrue = True ❷

mkFalse :: Bool ❶
mkFalse = False ❷
```

❶ `Bool` is the `Data Type`.

❷ `True` and `False` are the Data Constructors

`Bool` can be either `True` or `False` but NOT both. That's what makes it a ***Sum Type***. Sum Types are also called ***Unions*** because a union in Set Theory is an OR operation. The union of Set `A` and Set `B` contains elements that are contained in `A` OR contained in `B`.

Understanding this should help you see why we separate the different possible Values of `Bool` with `|`, which is commonly used as the OR symbol in many languages, i.e. `||`.

Let’s look at another `Sum Type`:

```
data FailureReason
  = InvalidSyntax
  | InvaidInput
  | AlreadyExists
  | NotFound
  | Other String
```

Here we have a Type to model a reason for failures with a catch-all case of `Other`. Notice that `Other` takes a `String` as a Parameter. Types can take Parameters just like Functions.

It turns out that the Data Constructor, `Other`, is a Function. Its ***implied*** Type is:

```
Other :: String -> FailureReason
```

This means that we can use `Other` the same way we use Functions because it is a Function.

All of the other Data Constructors for `FailureReason` are Values. Here are their ***implied*** types:

```
InvalidSyntax :: FailureReason
InvalidInput :: FailureReason
...
```

`FailureReason` is a bit limited though. Can you see why?

Notice that `Other` only takes a `String`. This may be fine for most situations but what if we want to use `FailureReason` elsewhere in our codebase where `String` isn’t enough to fully describe the `Other` case?

Let’s make a more Flexible version of `FailureReason`:

```
data FailureReason' a ❶
  = InvalidSyntax'
  | InvaidInput'
  | AlreadyExists'
  | NotFound'
  | Other' a
```

❶ We added `'` to the end of all of the names to distinguish them from our old definition.

Now we’ve introduced the ***Type Variable***, `a`, into the definition. This means that `Other'` can still take a `String` if we want, but now it also can take any other Type:

```
failWithString :: FailureReason' String ❶
failWithString = Other' "Because this Function always fails"

type Error = { code :: Int, reason :: String }

failWithError :: FailureReason' Error ❷
failWithError = Other' { code: -123, reason: "Because of reasons"}
```

❶ We pass `String` to the `FailureReason'` Type which unifies `a` with `String`. This means that `Other'` must take a `String`.

❷ Here `a` unifies with `Error` meaning `Other'` expects an `Error`.

Polymorphic vs Monomorphic

Let’s look at our two definitions:

```
data FailureReason ❶
  = InvalidSyntax
  | InvaidInput
  | AlreadyExists
  | NotFound
  | Other String

data FailureReason' a ❷
  = InvalidSyntax'
  | InvaidInput'
  | AlreadyExists'
  | NotFound'
  | Other' a
```

❶ This is ***Monomorphic*** because `FailureReason` takes no Type Parameters (***Mono*** means ***one*** and ***morph*** means ***shape***). Our `Other` Data Constructor is also Monomorphic since it can only take one shape or Type, viz. `String`.

❷ This is ***Polymorphic*** because `FailureReason'` takes a Type Parameter, `a` (***Poly*** means ***many*** and ***morph*** means ***shape***). Our `Other'` Data Constructor is Polymorphic since it can take many shapes or Types, e.g. `String`, `Error`, etc.

Monomorphic Types are always in uppercase, e.g. `String`, whereas, Polymorphic Types are always in lowercase, e.g. `a`.

Product Types

Let’s look at a simple Type:

```
data Triplet a b c = Triplet a b c
```

Here we have a 3 Polymorphic Type Parameters, a, b and c.

The Data Constructor, Triplet has the implied Type:

```
Triplet :: a -> b -> c -> Triplet
```

Let's create a Triplet:

```
type StringStats = Triplet String Int Int

getStats :: String -> StringStats
getStats s = Triplet s (length s) (vowelCount s) ❶ ❷
```

❶ The first slot of Triplet contains our String. The second slot contains the length and the third the number of vowels.

❷ vowelCount is an imagined Function whereas length is a Function in the Data.String.CodeUnits module.

StringStats simultaneously contains 1 String and 2 Integers. That's what makes it a ***Product Type***. It contains a String AND an Int AND another Int. In sets, this corresponds to Intersection. The intersection of Set A and Set B contains elements that are contained in A AND contained in B.

The trouble with Triplet is that we can mix up the 2 Ints. There are ways to fix this with newtype which we will look at later, but there's an even better way.

We can change StringStats to use a Record instead of using the potentially ambiguous Triplet:

```
data StringStats = StringStats ❶
  { string :: String ❷
  , length :: Int ❸
  , vowelCount :: Int ❹
  }
```

❶ StringStats is no longer a Type Alias, but it's a proper Data Type. This change is NOT a requirement to make a Record in PureScript (it is in Haskell). We could've left it a Type Alias.

❷ Holds the same as the first slot of Triplet.

❸ Holds the same as the second slot of Triplet.

❹ Holds the same as the third slot of Triplet.

`StringStats` is not ambiguous like `Triplet` is.

Isomorphic

Comparing our `Triplet` to `StringStats`:

```
data Triplet a b c = Triplet a b c

data StringStats = StringStats
  { string :: String
  , length :: Int
  , vowelCount :: Int
  }
```

`StringStats` and `Triplet` both contain the same information. The big difference is the fact that `Triplet` is more flexible since it can take any types `a`, `b` and `c`, whereas `StringStats` takes very specific types `String`, `Int` and `Int`:

They're almost the same, but let's work to make them closer. First, let's make a specialized version of `Triplet`:

```
data StringTriplet = StringTriplet String Int Int

data StringStats = StringStats
  { string :: String
  , length :: Int
  , vowelCount :: Int
  }
```

Now `StringTriplet` and `StringStats` have exactly the same types. That means we could use either Type interchangeably. There is a formal definition for this:

> Two Types, T1 and T2, are ***Isomorphic*** (***Iso*** means ***equal*** and ***morph*** means ***shape***) if a Function can be written from T1 to T2 and from T2 to T1 without any loss of information.

Let's see if our new types are Isomorphic by trying to write lossless conversion Functions between them:

```
from :: StringTriplet -> StringStats
from (StringTriplet string length vowelCount) = ❶
  StringStats ❷
  { string: string
  , length: length
  , vowelCount: vowelCount
  }

to :: StringStats -> StringTriplet
to (StringStats { string, length, vowelCount }) = ❸
  StringTriplet string length vowelCount ❹
```

❶ `from's` Parameter is destructured using ***Pattern Matching*** giving us access to the elements of the Product Type, `StringTriplet` (more on this later).

❷ We construct `StringStats` from the parts we destructured from `StringTriplet`.

❸ `to` also uses Pattern Matching to access fields of the Record in `StringStats` using a slightly different syntax since it's destructing a `Record`.

❹ We construct `StringTriplet` from the parts we destructured from `StringStats`.

Don't worry, we'll take a deeper dive into Pattern Matching soon enough. But the point of this code is to prove that we can write Functions to convert back and forth between `StringTriplet` and `StringStats` with NO information loss.

These Funtions prove that `StringTriplet` and `StringStats` are Isomorphic.

Inhabitants

The ***NO information loss*** part of Isomorphisms is very important. At first glance, you might think `String` and `Int` are Isomorphic since any `Int` can be converted into a `String`. But not any `String` can be converted into an `Int`.

Types are similar to Sets, except where Sets have Elements, Types have Inhabitants. And there are far more Inhabitants of `String` than `Int` even though they both have an Infinite number of Inhabitants.

This gets into the mind-bending concept of varying sizes of Infinity, but we can think of it simply by realizing that every `Int` has a string representation in the `String` Type, but `String` has additional inhabitants that don't have any digits, e.g. `""`, `"·_·"`, `"abc"`, etc.

The number of inhabitants in `String` is greater than the number in `Int`, which means that they cannot be Isomorphic.

Let's imagine a Type that only contains the numbers `42` and `79` and let's call it `TwoNum`. Because this has two inhabitants, it makes it Isomorphic to `Boolean` since it also has two inhabitants, `true` and `false`.

How we map between `TwoNum` and `Boolean` can be totally arbitrary. We could simply just make a lookup table that's used by our `to` and `from` Functions:

```
TwoNum    Boolean
------    -------
42        false
79        true
```

With a lookup table, we can always map back and forth between any two types as long as they have an equal number of inhabitants.

Any two types with an equal number Inhabitants are Isomorphic.

Let's look at Sum Types (Coproducts) and calculate their inhabitants:

```
data Variant = This | That | TheOther
```

It's easy to see that `Variant` has 3 inhabitants, `This`, `That` and `TheOther`.

Let's try another:

```
data DualResult = First Boolean | Second Variant
```

`DualResult` has 2 Data Constructors, `First` and `Second`, but that doesn't automatically mean that it has 2 inhabitants.

Since both `First` and `Second` have Type Parameters, we must take those into consideration. `First` can take a `Boolean` which has 2 inhabitants and `Second` takes a `Variant` which has 3 inhabitants. So let's enumerate all possible values of `DualResult`:

```
First true
First false
Second This
Second That
Second TheOther
```

Notice each Data Constructor takes a Parameter of Type `Boolean` (`2` inhabitants) OR `Variant` (`3` inhabitants). There's that `OR` again like we saw in Unions.

Taking all of this into consideration, it should be clear that `DualResult` has `2 + 3` inhabitants. This addition operation to calculate inhabitants is why we call it a `Sum Type`.

Let's see if this same logic holds with Product Types:

```
data BooleanVariant = BooleanVariant Boolean Variant
```

Be careful not to confuse the Type, `BooleanVariant` (lefthand side), with the Data Constructor, `BooleanVariant` (righthand side). Here are all of the possible values of the Type `BooleanVariant`:

```
BooleanVariant true This
BooleanVariant true That
BooleanVariant true TheOther
BooleanVariant false This
BooleanVariant false That
BooleanVariant false TheOther
```

Notice each Data Constructor has both a `Boolean` AND a `Variant`.

With a Product Type, we have to consider all combinations of these types, i.e. all combinations of the 2 inhabitants from `Boolean` paired with the 3 inhabitants from `Variant`, or `2 · 3` inhabitants. This multiplication operation is why we call this a `Product Type`.

When the Types are ORed as in `DualResult` we ADD (Sum) and when the types are ANDed as in `BooleanVariant` we MULTIPLY (Product).

But what happens when they're combined:

```
data Combo = Dual DualResult | BoolVar BooleanVariant
```

The `|` tells us that we're adding the inhabitants of each side. So we have `(2 + 3) + (2 · 3)` or `5 + 6` inhabitants because `DualResult` has 5 and `BooleanVariant` has 6.

So far we've been calculating inhabitants for Monomorphic Types. How do we calculate the inhabitants for Polymorphic Types?

```
data Sometimes a = HaveIt a | DoNotHaveIt
```

`Sometimes` represents an `a` sometimes, i.e. we may or may not have an `a`.

The number of inhabitants are `a` for `HaveIt` and `1` for `DoNotHaveIt`. That means that the number of inhabitants for `Sometimes` is `a + 1`.

Here `a` stands for ***the number of inhabitants of a***.

If a is Boolean as in:

```
type SomeBool = Sometimes Boolean ❶
```

❶ a unifies with Boolean.

then the number of inhabitants of SomeBool is 2 + 1. We got this by replacing the a in a + 1 with 2 for the number of inhabitants in Boolean.

New Types

We’ve just learned how we can create our own Types to model our problem domain. Next, let’s look at how we might model some simple problems.

Let’s write a simple Function to format a name:

```
fullName :: String -> String -> String -> String
fullName first middle last = first <> " " <> middle <> " " <> last ❶
```

❶ The operator <> appends strings to each other.

Forgetting for now that our Function will have 2 spaces between the first and last name if the middle name is an empty string, can you see anything else wrong with our Function?

HINT: Look at the Types.

Imagine we call our Function like:

```
fullName "Smith" "Jay" "John"
```

We accidentally put the last name first. How did that happen?

Looking at just the Type Signature:

```
fullName :: String -> String -> String -> String
```

we can see that it’s not very helpful regarding the order of our Parameters. So, let’s make it so we can read this better by creating Type Aliases:

```
type FirstName = String
type MiddleName = String
type LastName = String

fullName :: FirstName -> MiddleName -> LastName -> String
fullName first middle last = first <> " " <> middle <> " " <> last
```

Now, when we look at the Type Signature, we will be able to tell how we should be calling this, greatly reducing our chance for errors:

```
fullName "Smith" "Jay" "John"
```

Well, we did it again! We made the same mistake. The Type Aliases are helpful but only if we the programmer read them. How can we make this better?

Ideally, we'd like our program to fail to compile if we get these in the wrong order. So let's create ***unique*** Types for each Parameter in our Function:

```
data FirstName = FirstName String ❶
data MiddleName = MiddleName String
data LastName = LastName String
data FullName = FullName String

fullName :: FirstName -> MiddleName -> LastName -> FullName
fullName (FirstName first) (MiddleName middle) (LastName last) = ❷
  FullName (first <> " " <> middle <> " " <> last) ❸
```

❶ We wrap all of our `Strings` in unique Types.

❷ We destructure all of our Parameters using Pattern Matching.

❸ The final result is wrapped in `FullName`.

Now let's call our Function with our new Types:

```
-- COMPILER ERROR!
fullName (LastName "Smith") (MiddleName "Jay") (FirstName "John")
```

We called `fullName` with the Parameters in the wrong order again. But this time, we get a compiler error because even though `FirstName` and `LastName` both take `Strings`, they are not the same Type anymore.

So when we accidentally passed `LastName "Smith"` where the Function was expecting a `FirstName`, the compiler caught the `Type Mismatch` error, protecting us from ourselves.

This technique just ***wraps*** a Type inside of another Type. In our case, we wrapped our String in another Type which made it unique. We essentially made a new Type for each String in our original Function.

PureScript has a special keyword for this approach called `newtype`:

```
newtype FirstName = FirstName String ❶
newtype MiddleName = MiddleName String
newtype LastName = LastName String
newtype FullName = FullName String

fullName :: FirstName -> MiddleName -> LastName -> FullName
fullName (FirstName first) (MiddleName middle) (LastName last) =
  FullName (first <> " " <> middle <> " " <> last)
```

❶ `data` has been replaced with `newtype`

The `newtype` keyword tells the compiler that we're just making a new Type for another Type, e.g. `FirstName` is a new Type for `String`. That means that the compiler can do some optimizations if it knows that it's just a simple wrapper.

But it can only do these optimizations if certain restrictions are imposed:

- `newtypes` must only have 1 Data Constructor
- The Data Constructor can only take 1 Parameter

The Type `FullName` has a single Data Constructor, also called `FullName` that takes a single Type Parameter, `String` as does `MiddleName`, `LastName` and `FullName`. This is why we could replace `data` with `newtype`.

There are additional features that makes working with `newtypes` convenient. We will leverage these features after we learn about ***Typeclasses***.

3.1.4. Common Library Types

PureScript implements many commonly used Types in its libraries. We're going to explore a few of them here.

`Void`

In Set Theory, there's the concept of an ***Empty Set*** that has no elements.

In Type Theory, there's the concept of ***Void Type***, which is a Type with Zero Inhabitants. In PureScript (and Haskell) this Type is called `Void`.

Of all the Types we'll explore, `Void` is probably the least used.

You might be wondering how you would define a Type that has no inhabitants. If we try, we might write:

```
data NoInhabitants = NoInhabitants
```

Unfortunately, this has 1 inhabitant, viz. `NoInhabitants`.

There is one possible solution. We can put this Type definition in a Module and then only export out the Type and NOT the Data Constructor. This way no one can construct a `Void`.

But there's another, much more clever and simpler solution:

```
data Void = Void Void
```

At first glance, this may be confusing, so let's break it down from Left to Right:

- The first `Void` is the Data Type.
- The second `Void` is the Data Constructor.
- The third `Void` is the Type Parameter to the `Void` Data Constructor.

So to create a `Void` we have to use the `Void` Data Constructor and pass it a `Void`:

```
v :: Void
v = Void (???)
```

The `???` needs to be of type `Void`, which we can create using the `Void` Data Constructor:

```
v :: Void
v = Void (Void (???))
```

But our next Data Constructor also needs a `Void`, which we can create using the `Void` Data Constructor and so on:

```
v :: Void
v = Void (Void (Void (Void (Void(Void(Void(...)))))))
```

It is impossible to construct a `Void` since its only Data Constructor requires a `Void`. We have a chicken and egg problem.

The technical term for this is a ***Recursive Definition*** because the definition refers to itself.

Except, in this particular case, it's an ***Infinitely Recursive Definition*** because there is no base-case to terminate the recursion.

Let's look at a made-up, Infinitely Recursive Definition:

```
data NeverEnding = NeverEnding NeverEnding
```

Now, we add a base-case to turn this into just a Recursive Definition:

```
data NeverEnding = NeverEnding NeverEnding | TheEnd ❶
```

❶ `TheEnd` is our base-case since it ***terminates*** the Recursive Definition.

This definition is no longer Infinitely Recursive thanks to the newly add Data Constructor, `TheEnd`. And because of that, we can actually write down something of type `NeverEnding`:

```
actuallyEnds :: NeverEnding
actuallyEnds = NeverEnding (NeverEnding (NeverEnding TheEnd))
```

The first and second `NeverEnding` Data Constructors needed something of Type `NeverEnding` and got it by using the `NeverEnding` Data Constructor.

But the third and final `NeverEnding`, which also needs something of Type `NeverEnding`, got `TheEnd` instead, which terminates our construction.

`Unit`

In Set Theory, there's the concept of a ***Unit Set*** that has exactly 1 element.

In Type Theory, there's the concept of ***Unit Type***, i.e. a Type with only One Inhabitant and in PureScript this Type is called `Unit`.

The only inhabitant of `Unit` is called `unit`.

`Unit` appears often in code with Effects. Our `main` Function in our `Hello World` example has the following signature:

```
main :: Effect Unit
```

In fact, the entry point to all PureScript programs will always have this Type Signature.

`Maybe`

Anyone who has coded in C, Java, Javascript and a host of other languages where values can be NULL knows all too well the pain of dealing with these cases. It seems that NULL values have haunted programmers since the beginning of time. Well, at least since the beginning of NULL values.

It turns out that PureScript solves this problem by having NO NULLs. You may wonder how you handle optional values without NULLs. Well, it's done with the `Maybe` Type.

Here's its definition and an example of how it could be used:

```
data Maybe a = Just a | Nothing ❶

data Person = Person ❷
  { name :: String
  , birthdate :: Date
  , deathdate :: Maybe Date ❸
  }
```

❶ Maybe takes a single, Polymorphic Parameter, `a`, which can be ANY Type.

❷ Person is a `Record` with 3 fields.

❸ `deathdate` is a `Maybe Date` since the `Person` may still be living. In this case, `Maybe's` Type Parameter, `a`, unifies with `Date`.

To set `deathdate` for an existing `person` we'd do the following:

```
import Data.Date (canonicalDate)

person :: Person
person = Person
  { name: "Joe Mama"
  , birthdate: canonicalDate 1962 10 2 ❶
  , deathdate: Nothing ❷
  }

-- somewhere else in our code
deadPerson = person { deathdate = Just today } ❸ ❹
```

❶ We set `birthdate` by calling `canonicalDate`, which takes a `year`, `month` and `day` and returns a `Date`.

❷ When this record is constructed, there is no `deathdate`, so we construct a `Maybe Date` by using the constructor `Nothing`.

❸ Later in our code, we set the `deathdate` by using the Data Constructor `Just`. But we cannot simply use `Just` by itself. `Just` requires a single Parameter of Type `a`, which, in this case, is `Date`. Therefore, we must give it a `Date` and we do so by give the Data Constructor a single Parameter, `today`, which we'll pretend is a `Date`.

❹ We're using the Update `Record` syntax, hence the use of `=` and NOT `:`.

This is all fine and good, but you may still be wondering how this is better than NULL. This can best be answered by looking at an example.

First in Javascript:

```
const x = 10;
const y; ❶
const add = (x, y) => x + y
const z = add(x, y) // NaN ❷
```

❶ We did not initialize `y`.

❷ Here we try to use `y` in a computation and wind up with `NaN`. Now any computation that uses `z` will be `NaN`.

Now in PureScript:

```
add :: Int -> Int -> Int
add x y = x + y

useAdd :: Int
useAdd =
  let x :: Int
      x = 10

      y :: Maybe Int
      y = Nothing ❶
  in
  add x y -- COMPILER ERROR!! ❷
```

❶ We have no value for `y` and it’s EXPLICITLY defined to be a `Maybe Int`.

❷ The compiler complains because we tried to add an `Int` with a `Maybe Int`.

The compiler saves us from using an incompatible Type. So how do we fix this?

We could try to make the Types compatible:

```
add :: Int -> Int -> Int
add x y = x + y

useAdd :: Int
useAdd =
  let x :: Int
      x = 10

      y :: Int ❶
      y = Nothing  -- COMPILER ERROR!!
  in
  add x y ❷
```

❶ If we try to make `y` an `Int` but set it to `Nothing`, the compiler will complain since our types don't match (`Int ≠ Maybe Int`)

❷ The compiler doesn't complain here because we defined `y` to be of Type `Int` which is fine for adding to another `Int`.

Once again the Type system saves us from assigning a `Maybe Int` to an `Int`. But how do we fix this now?

We could just initialize `y` to some Value:

```
add :: Int -> Int -> Int
add x y = x + y

useAdd :: Int
useAdd =
  let x :: Int
      x = 10

      y :: Int
      y = 42
  in
  add x y
```

Now everything works. But this example is too simplistic. What would happen if `useAdd` got passed `y`:

```
add :: Int -> Int -> Int
add x y = x + y

useAdd :: Maybe Int -> Int ❶
useAdd y = ❶
  let x :: Int
      x = 10
  in
  add x y  -- COMPILER ERROR!! ❷
```

❶ We now get passed `y`, so our Type Signature and Parameter list reflect that.

❷ And we get a compiler error just like before since we cannot add `Int` to a `Maybe Int`.

We haven't fixed anything just yet, but now this problem is more realistic. So now let's fix it by making `useAdd` return a `Maybe Int` to reflect the fact that we MAY fail:

```
add :: Int -> Int -> Int
add x y = x + y

useAdd :: Maybe Int -> Maybe Int ❶
useAdd y' = ❷
  let x :: Int
      x = 10
  in
  case y' of ❸
    Just y -> Just (add x y) ❹
    Nothing -> Nothing ❺
```

❶ Our Type Signature changed to return a `Maybe Int`.

❷ Our Parameter name changed to `y'` so it won't conflict with `y` later.

❸ We are using a `case` expression in PureScript (more on this later).

❹ We are Pattern Matching with the `Just y` where `y` will get bound to the Value ***inside*** the `Maybe`, which is an `Int` and can be safely used in `add x y`.

❺ If we don't get a `y`, i.e. it's `Nothing`, then all we can do is return `Nothing`.

Notice how the `Maybe` Type forced us to handle the fact that `y'` may not exist. The compiler hounded us to get our types in order.

At first, you may curse the compiler for this, I know I did. But over time, you'll begin to love the fact that all possible errors have been "handled". It's still possible to handle this case badly, but at least you are forced to think about each and every one of them.

In the Javascript case, our Value became a `NaN` which is Isomorphic to `Nothing`. The difference being that the PureScript compiler enforces the use of `Maybe`. And if we use best practices, `Maybe` will show up in our Type Signatures, which will inform us and others to the fact that our Function may fail.

So the Semantics of `Maybe` is something that can fail. In our example, `useAdd` cannot use `add` in the absence of one of the `Ints`.

There are many times were `Maybe` comes in handy:

- Search `String` for a `Char` or another `String`.
- Get a Value from `Map` (a Key/Value store) using a key.
- Get a Value from an `Array` at a particular index.

Each of these Functions have a single ***obvious*** failure, `Char` or `String` not found, key not found or index out of bounds.

But what if that's not the case or our Function can fail for multiple reasons? How do we want return an explicit failure reason?

Either

The `Either` Type is another failure Type, like `Maybe`, except it has the added benefit of having a reason for the failure:

```
data Either a b = Left a | Right b
```

Here Type `a` is used by the `Left` Data Constructor and Type `b` is used by the `Right` Data Constructor. To get a better sense of this, let's look at an example:

```
data QueryError ❶
  = DatabaseConnectionError String ❷
  | InvalidQuery
  | NotFound

query :: Query -> Either QueryError Int ❸
```

❶ `QueryError` models the different errors we can encounter.

❷ `DatabaseConnectionError` takes a `String` which contains more information about the connection failure.

❸ `query` takes a `Query` (not defined here) and either returns a `QueryError` or an `Int`.

So how would we create the different errors:

```
Left (DatabaseConnectionError "The connection was refused by the server")
Left InvalidQuery
Left NotFound
```

With Either, the left side is ALWAYS the error type. This isn't just a convention. Later, we'll see why this is the case, but for now, just remember Left is the error and Right is success. I always think of Right as being the right result and anything else is wrong or an error.

Here's how we represent a successful result for query:

```
Right 42
```

Let's revisit the definition of Either:

```
data Either a b = Left a | Right b
```

Either has 2 Type Parameters, a and b. What this means is that the Type on the left side CAN be different than the Type on the right.

But it doesn't have to be:

```
Either Int Int
```

This would mean that it's going to return an Int or an Int which is perfectly valid. Just because a and b are different names only says that they ***can*** be different Types, but they're completely independent from each other. Therefore a can be an Int and so can b.

But how do we tell the difference between an Int that's an error and an Int that's a success?

We use the case expression and Pattern Matching:

```
case result of ❶
  Left errorCode -> "The error code is: " <> show errorCode ❷ ❹
  Right count -> "The number of rows returned is: " <> show count ❸ ❹
```

❶ result is NOT an Int, it's an Either Int Int.

❷ We match on Left to extract errorCode.

❸ We match on Right to extract the count.

❹ We have to use show with errorCode and count to convert them to Strings so that they can be appended with the rest of the message.

You can think of `Left` and `Right` as ***Tagging*** the `Int` so that we can tell whether it represents an error or a success. This is why `Sum Types` are also known as `Tagged Unions`.

Maybe vs Either

`Maybe` and `Either` can both represent error results, but only `Either` gives us a reason. `Maybe`, on the other hand, can represent an optional Value.

Just for fun, let's look at the inhabitants of both:

```
data Maybe a = Just a | Nothing

data Either a b = Left a | Right b
```

Remember that `Maybe` is a `Sum Type`, so we need to add the inhabitants of all of the Data Constructors. The inhabitants of `Maybe` is `a` for `Just a` and `1` for `Nothing`, therefore the number of inhabitants is `a + 1`.

And for Either, which is also a `Sum Type`, we have `a` for `Left` and `b` for `Right` or `a + b` inhabitants.

```
Maybe:    a + 1
Either:   a + b
```

That was a mildly interesting exercise, but there's actually something more interesting if we try to make `Maybe` and `Either` have the same inhabitants, which would make them `Isomorphic`.

There's 2 ways we can do that.

First, we choose a Type for `b` such that it has only 1 inhabitant:

```
Maybe:    a + 1
Either:   a + 1     (b = 1)
```

One thing to note here is that an `a` in the `Maybe` Type is independent of an `a` in the `Either` Type. So, an easy way to make them have the same number of inhabitants, is to simply choose the same Type for both `a's`. We also could've picked 2 different Types with equal Inhabitants.

Let's choose `Boolean` for both `a's` and `Unit` for `b` (we need `b` to have 1 inhabitant):

```
Maybe Boolean ≅ Either Boolean Unit ❶
```

❶ The symbol `≅` stands for `is Isomorphic to`.

Notice that our `Either` has an error Value but no success Value. While they're technically Isomorphic, `Either Boolean Unit` doesn't make sense with the Semantics of `Maybe Boolean`. That's because the `Boolean` in `Maybe` is the success Type, whereas the `Boolean` in `Either` is the error Type.

The second way to make them Isomorphic is to swap the `Either` types:

```
Maybe Boolean ≅ Either Unit Boolean
```

Now our `Either` has a success Value but no error Value, which is exactly like `Maybe`. It too only has a success Value. Semantically, this Isomorphism makes sense. `Boolean` in both types is the success Type.

So we have 3 inhabitants for each of the following:

```
Maybe Boolean ≅ Either Boolean Unit ≅ Either Unit Boolean
```

Let's list them:

```
Maybe Boolean
-------------
Just true
Just false
Nothing

Either Boolean Unit
-------------------
Left true
Left false
Right unit

Either Unit Boolean
-------------------
Left unit
Right true
Right false
```

While this was slightly more interesting, you may be wondering what this has to do with using these Types. It turns out that there are times we want to convert between `Maybe` and `Either` and when they're Isomorphic we can easily do this:

```
to :: Maybe Boolean -> Either Unit Boolean
to (Just b)  = Right b
to Nothing   = Left unit

from :: Either Unit Boolean -> Maybe Boolean
from (Right b) = Just b
from (Left _)  = Nothing ❶
```

❶ We can use `_` to represent a Variable that we don't care about naming. This is a common pattern when working with `unit`.

But, let's be real, we're never going to have an `Either` where one of the Type Parameters is of Type `Unit`. We're just going to use a `Maybe`.

So how do we convert between them when they're not Isomorphic?

Looking at `to` and `from` above gives us some clues.

When we went from `Maybe` to `Either`, we had to make up data for the `Nothing` case. We did that by using `unit`. When we went from `Either` to `Maybe`, we threw out data for the `Nothing` case. We threw out the `unit`.

If we want to convert between ***any*** `Maybe` and `Either`, we'll need to do something similar:

```
hush :: Either a b -> Maybe b
```

Notice there is no `a` in the `Maybe` result. That should tell us that `hush` "throws away" the error Type on the left side of the `Either`.

Let's look at the rest of the Function:

```
hush :: ∀ a b. Either a b -> Maybe b ❶
hush (Left _)  = Nothing ❷ ❹
hush (Right x) = Just x ❸ ❹
```

❶ When defining Polymorphic Type Parameters for Functions, you must explicitly define them using the keyword `forall` or you can optionally use the symbol, `∀`. I use the symbol because it's easier to read.

❷ `_` is used since we aren't going to use the `a` from `Left`.

❸ We destructure the `Right` to get to the `x`, which is of Type `b`, and then we construct a `Maybe` using `Just`.

❹ In PureScript, we can write different versions of Functions. The compiler matches the Parameters passed in to determine which version to call. Unlike languages that support overloading, they MUST have the same Type Signatures.

`hush` is in the library `package.either` in module `Data.Either`.

To go the other direction, i.e. from a `Maybe` to an `Either`, we need to provide the Value for the left side of the `Either` since `Maybe` only has `Nothing` for its error case:

```
note :: ∀ a b. a -> Maybe b -> Either a b ❶
note err Nothing = Left err ❷
note _ (Just x) = Right x ❸ ❹
```

❶ The first Parameter is of Type `a`, which is the same Type as the `a` side of the `Either`.

❷ In this case, we have `Nothing` and so we must use the provided error Value, `err`, to construct a `Left`.

❸ We don't need the error Value here since this case is the success case, i.e. the right side of `Either`. So we just use _ as our "don't care" Variable name.

❹ We destruct the `Just` to get at the `x` and then immediately wrap it in `Right`.

`note` is also a library Function that can be found in the same module as `hush`, viz. `Data.Either`.

Tuple

A `Tuple` is a `Product Type` that contains 2 things. Here's its definition:

```
data Tuple a b = Tuple a b
```

In Math, a Tuple is written as `(x, y)`. Haskell and Elm use the same syntax for Tuples that's used in Math. In other programming languages, this is often called a pair.

A `Tuple` is useful when you need to create a pair of things that are related in some way.

In Math, we used tuples for `x-y coordinates` on a 2D graph, e.g. `(5, -1)`. To fully describe a point in a plane, we need both the `x` and `y` coordinates. It makes sense to represent them as a tuple:

```
type Point = Tuple Int Int
```

Another use case for `Tuple` is when you want to return 2 values back from a Function:

```
splitPosAndNeg :: Array Int -> Tuple (Array Int) (Array Int)
```

`splitPosAndNeg` will take an `Array` of `Int` and return a `Tuple` where the first position of the `Tuple` contains an `Array` of Negative Numbers and the second contains an `Array` of Positive.

Here's how we might use it:

```
let Tuple pos neg = splitPosAndNeg [1, -2, 3, -4, 5, -6]
```

We are using destructuing here to assign pos to the first of the Tuple and neg to the second.

We can use library Functions to do the same:

```
let split = splitPosAndNeg [1, -2, 3, -4, 5, -6]
    pos = fst split ❶
    neg = snd split ❷
```

❶ fst gets the first Value of a Tuple.

❷ snd gets the second Value of a Tuple.

fst and snd are in the package purescript-tuple in module Data.Tuple.

Either vs Tuple

Earlier, we made Monomorphic versions of Maybe and Either such that they were Isomorphic.

In other words, we came up with concrete types for the Polymorphic Type Parameter a for Maybe and the Polymorphic Type Parameters a and b for Either to make Maybe and Either Isomorphic.

Can we do the same with Tuple and Either?

The real question that we're asking is can we take a Product Type such as Tuple Isomorphic to a Sum Type such as Either. Remember that a Product Type is an ***AND*** and a Sum Type is an ***OR***.

Tuple a b means that this Tuple has an a ***AND*** a b, whereas Either a b means that this Either has an a ***OR*** a b.

To answer this, let's first look at their inhabitants:

```
Tuple:  = a * b
Either: = a + b
```

Remember that Tuple and Either are Isomorphic when the number of inhabitants are equal, meaning that the following must be true:

```
a * b = a + b
```

There's only 2 times that the addition of 2 numbers and multiplication of those same 2 numbers are equal:

```
0 * 0 = 0 + 0     (a, b = 0)
2 * 2 = 2 + 2     (a, b = 2)
```

Let's write the corresponding Types:

```
Tuple Void Void ≅ Either Void Void
Tuple Boolean Boolean ≅ Either Boolean Boolean
```

Although we've made a `Product Type` and a `Sum Type` Isomorphic, with these 2 cases, the Types are pretty useless as far as Types go.

Because of this, conversions between `Product Types` and `Sum Types` are rarely done, if ever.

Case in point, there are no library Functions to convert from `Tuple` to `Either` or vice versa.

List

`Array` implementations are backed up by Javascript Arrays for efficiency and you'll most likely use `Arrays` more than `Lists`, but there are things you cannot do with Arrays as we'll see in Pattern Matching.

And learning how `Lists` work in PureScript will help you to understand how to work with `Lists` in Haskell where they are used much more often than their array counterparts.

We will also write many of the `List` library Functions as exercises that will get you used to programming functionally and recursively.

Here is the Recursive Definition of a `List` and its corresponding operator:

```
data List a = Cons a (List a) | Nil ❶

infixr 6 Cons as : ❷
```

❶ A `List` of `a's` is either the empty list, `Nil` or an `a` that is `Cons-ed` to the head of another `List` (more on this in a minute).

❷ The `:` is an infix operator for the `Cons` Data Constructor. It has precedence 6 and is Right-Associative, hence the `r` at the end of `infixr` (more on this in a minute).

The `Cons` operation puts a single Value onto the head of a `List`.

We can construct a `List` with the `Cons` operator:

```
nums :: List Int
nums = 1 : 2 : 3 : Nil
```

Remember the operator, `:`, is Right-Associative, which means it associates to the right making the previous example equivalent to the following:

```
nums :: List Int
nums = (1 : (2 : (3 : Nil))) ❶
```

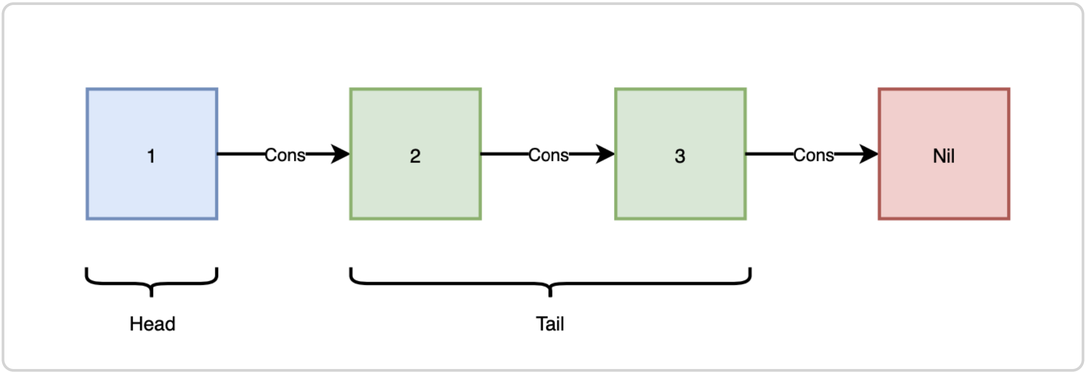
❶ The redundant Parentheses show the Right-Associativity of the operator. Notice how the Parentheses collect on the RIGHT. These Parentheses show that the order of operations move from right to left. First `3` is `Cons-ed` with `Nil`. Then `2` is `Cons-ed` with that resulting `List` and finally, `1` is `Cons-ed`.

We could have constructed the list with just the `Cons` Data Constructor instead of the operator:

```
nums :: List Int
nums = Cons 1 (Cons 2 (Cons 3 Nil)) ❶
```

❶ This syntax helps us see the Recursive Definition in action but is more difficult to read and reason about. That's where the `:` operator is helpful.

`Lists` are effectively ***Linked Lists***. Here is a pictoral view of our list:



The `head` of the list is the first item in the list. The `tail` is everything except the `head`.

In the module `Data.List`, there are Functions for getting the `head` and `tail` of a `List`:

```
head :: ∀ a. List a -> Maybe a
head Nil      = Nothing ❶
head (x : _) = Just x ❷

tail :: ∀ a. List a -> Maybe (List a)
tail Nil        = Nothing ❸
tail (_ : xs) = xs ❹
```

❶ If the `List` is empty, then we cannot get the `head` and therefore return `Nothing`.

❷ Here we Pattern Match against the `Cons` operator to get the `head` of the `List`. We don't care about naming the `tail` and so use `_`.

❸ If the `List` is empty, then there is no tail.

❹ We Pattern Match against the `Cons` operator to extract the `tail` and return it. We don't care about the `head` and call it `_`.

We'll see how this Pattern Matching works next.

`Cons` has its origins in Lisp, a Programming Language from the 1950s. Lisp stands for **Lis**t **P**rocessor. In Lisp, `cons` is the ***Constructor*** for lists.

3.2. Pattern Matching

In our study of Types, we glossed over the idea of ***Pattern Matching*** in PureScript. Let's take a deeper dive.

Pattern Matching matches values based on both their Shape and their Value, e.g. `Just 10` and `Just 5` have the same Shape but different Values, whereas `Nothing` and `Just "xyz"` have different Shapes.

3.2.1. Case Expression

A simple example:

```
import Data.Maybe (Maybe(..)) ❶

isNothing :: ∀ a. Maybe a -> Boolean ❷
isNothing m = case m of ❸
  Nothing -> true ❹
  _       -> False ❺
```

❶ `Maybe` is in the `Data.Maybe` module. The `(..)` after the `Maybe` say that in addition to importing the Type `Maybe`, we want to include ALL of its Data Constructors.

❷ Don’t forget the `∀ a`. Notice that `.` terminates the Type Variable definitions.

❸ We use `case` on `m` to do Pattern Matching on it.

❹ Our first case is to check for `Nothing`. The expression to the right of `->` is the Value of the Function if the pattern matches.

❺ The `_` is normally a dont-care variable and it has a similar semantic here. This is a catchall for any pattern, sort of a dont-care pattern.

There is another, more idiomatic, way to write this Function, which is to write it by defining multiple versions of the same Function but with different patterns for the input Parameters:

```
import Data.Maybe (Maybe(..))

isNothing :: ∀ a. Maybe a -> Boolean
isNothing Nothing = true ❶
isNothing _       = false ❷
```

❶ This version of `isNothing` will be called if the input Parameter matches `Nothing`.

❷ This version of `isNothing` will be called if any previous versions did not match.

Our patterns must cover all possible values of each input Parameter. This requirement is also imposed on the `case` expression.

The compiler will let you know if your patterns are not ***exhaustive***, i.e. you missed a possible case.

Pattern Matching starts with the first definition and proceeds to subsequent ones. Same for `case` pattern matching. Therefore, all catchall cases must be last.

The compiler will let you know if a case is ***unreachable***, e.g. you didn’t put a catchall last.

3.2.2. String Patterns

Let’s look at another example:

```
toString :: Boolean -> String
toString true  = "true"
toString _     = "false" ❶
```

❶ This catchall case is equivalent to `toString false`. We just chose to use `_` for brevity.

And the inverse Function:

```
fromString :: String -> Boolean
fromString "true" = true
fromString _      = false ❶
```

❶ This catchall case is NOT the same as the catchall in toString. What _ represents is any string that's NOT "true". So both "TRUE" and "false" will match, since pattern matching of Strings is case-sensitive.

3.2.3. Array Patterns

Arrays can only be Patterned Matched against ***fixed-length*** Arrays. So we can write an isEmpty Function using Pattern Matching:

```
isEmpty :: ∀ a. Array a -> Boolean
isEmpty [] = true ❶
isEmpty _ = false
```

❶ An empty Array literal is written as [].

We can also write:

```
multiplyTwo :: Array Int -> Maybe Int
multiplyTwo [x, y] = Just (x * y)
multiplyTwo _      = Nothing
```

Pattern Matching with Arrays is far less flexible than with Lists.

3.2.4. List Patterns

When we first looked at Lists, we saw two helper Functions, head and tail:

```
head :: ∀ a. List a -> Maybe a
head Nil     = Nothing
head (x : _) = Just x ❶

tail :: ∀ a. List a -> Maybe (List a)
tail Nil      = Nothing
tail (_ : xs) = Just xs ❶
```

❶ The Cons operator, :, can be used for Pattern Matching.

Idiomatic PureScript (and Haskell) uses a single character Variable, e.g. x, for the head of a List and that same variable with an appended s, i.e. xs, to represent the tail.

You can think of the head as a single x and the tail as a plural of x or xs.

Let's look at how we can use Cons with Pattern Matching:

```
head :: ∀ a. List a -> Maybe a
head Nil        = Nothing
head (Cons x _) = Just x ❶

tail :: ∀ a. List a -> Maybe (List a)
tail Nil         = Nothing
tail (Cons _ xs) = Just xs ❷
```

❶ We match just like we would any other Data Constructor with 2 Parameters. Here the first Parameter, x, is the head.

❷ Here the second Parameter of Cons, xs, is the tail.

Take a minute and compare this with head and tail that use the operator. Personally, I like the operator version better. It turns out that you will use the operator to Pattern Match far more often than Cons.

3.2.5. Array vs List

Pattern Matching an Array, requires you to know its exact size. You're also forced to deal with all the parts of the Array all at once:

```
addThree :: Array Int -> Maybe Int
addThree [x, y, z] = Just (x + y + z)
addThree _         = Nothing
```

With Lists, we can write Functions that deal with any length List and when we do, we can work with just the head and then recurse to continue with the tail of the list:

```
sum :: List Int -> Int
sum Nil      = 0 ❶
sum (x : xs) = x + sum xs ❷
```

❶ This Pattern Matches with an empty list, which has a zero length.

❷ We Pattern Match to extract both the head and tail. Then we add the head, x, to the sum of the tail, sum xs.

This Function is recursive, but don't worry if you're not really good with Recursion. We're going to write many Recursive Functions in the following chapters. It will become much easier.

The main point here is that with `Lists`, we don't have to know the size of the `List` and we don't have to deal with all of the elements of a `List` at once like we do with `Arrays`.

The limits on Pattern Matching with `Arrays` means that it's impossible to write the `sum` Function for `Array` using Pattern Matching:

```
sum :: Array Int -> Int
sum ???? -- Cannot write a Pattern here
```

We cannot match an `Array` of any size. There are many other solutions to this problem, one of which is using a `List` instead or possibly converting our `Array` to a `List` and using `List` Pattern Matching.

Later, when we learn about ***Folds***, we'll see the best solution for summing `Arrays`.

3.2.6. Record Patterns

Let's start with the following records:

```
type Address =
  { street :: String
  , city :: String
  , state :: String
  , zip :: Int
  }

type Employee =
  { name :: String
  , jobTitle :: String
  , yearsAtCompany :: Int
  , address :: Address
  }

type Company =
  { name :: String
  , yearsInBusiness :: Int
  , address :: Address
  }
```

Now let's look at a bunch of different ways to write a Function to check to see if the `Employee` is the `CEO`. First, the most naïve:

```
isCEO :: Employee -> Boolean
isCEO employee = if employee.jobTitle == "CEO" then true else false ❶
```

❶ Accessing a field of a `Employee Record` is done with `employee.jobTitle`.

The syntax for accessing `Records` in PureScript uses the same dot-notation that you see in many other languages, e.g. Javascript. But unlike Javascript, you cannot create an accessor at runtime.

Let's continue by removing the `if` expressions since it's redundant:

```
isCEO :: Employee -> Boolean
isCEO employee = employee.jobTitle == "CEO"
```

Now let's write this using Pattern Matching:

```
isCEO :: Employee -> Boolean
isCEO { jobTitle } = jobTitle == "CEO" ❶
```

❶ `{ jobTitle }` pattern matches the `jobTitle` field and binds it to a local variable with the same name.

When we use `{ jobTitle }`, this is called a ***Record Pun***.

Since we only care about the `jobTitle` in the `Employee` record, we only specified it. We could have specified other fields from that record:

```
isCEOAndGates :: Employee -> Boolean
isCEOAndGates { name, jobTitle } = jobTitle == "CEO" && name == "Bill Gates"
```

Let's write a Function to check the State of a `Company` for California:

```
isCalifornia :: Company -> Boolean
isCalifornia { address: { state } } = state == "CA" ❶
```

❶ This is how you Pattern Match nested records. `address` is of Type `Address`, which is a `Record`. Don't miss the `:` after `address`. That's easy to forget.

What would happen if we wanted Functions to check both the company and the employee's `state`:

```
isCompanyCalifornia :: Company -> Boolean
isCompanyCalifornia { address: { state } } = state == "CA"

isEmployeeCalifornia :: Employee -> Boolean
isEmployeeCalifornia { address: { state } } = state == "CA"
```

These two Functions are nearly the same. Imagine we have other Records with `Addresses` that we'd like to check their states. We'd have to repeat this boilerplate code over and over again.

There is a more general way to write these two Functions:

```
isCalifornia :: ∀ r. { address :: Address | r } -> Boolean ❶ ❷
isCalifornia { address: { state } } = state == "CA"
```

❶ We have to remember to define `∀ r`. Don't forget the `.` after the Variable definition.

❷ The Type `{ address :: Address | r }` says that we have a Record with AT LEAST a field named `address` of Type `Address`. The `| r` says that there MAY be other fields, but we don't care what they are.

What's great about this definition is that it'll match BOTH `Company` and `Employee` even though their other fields don't match. Those other fields are represented by `r` and since this definition is for all `r's`, i.e. `∀ r`, it'll match any set of fields.

The technical term for `r` is a ***Row Type***. More on this later.

There's one more important thing we can do when Pattern Matching Records and that's rename the field Parameter:

```
isCalifornia :: ∀ r. { address :: Address | r } -> Boolean
isCalifornia { address: { state: s } } = s == "CA"
```

Renaming is useful when a field name conflicts with another variable in scope, e.g. a Function name in the same module.

In our example, we can imagine that we renamed the field variable because there's a Function called `state` in the same module as our `isCalifornia` Function.

We can also Pattern Match to specific values of fields in Records:

```
isCalifornia :: ∀ r. { address :: Address | r } -> Boolean
isCalifornia { address: { state: "CA" } } = true ❶
isCalifornia _  = false ❷ ❸
```

❶ Pattern Match with `state` equals `"CA"`.

❷ With this approach, we're forced to provide other versions of the same Function. In this particular case, this approach is more verbose.

❸ Our catchall case is always `false`.

3.3. Logical Control

There are only a few ways to do logical decision making in PureScript:

- `if-then-else` expression
- `case` expression
- Pattern Matching
- Guards

3.3.1. If-Then-Else Expression

Notice, how I used the word ***expression*** when describing `if-then-else`. That's because it's NOT a statement as in many languages. An `if` must always have an `else` part. To understand why, let's look at a simple example:

```
keepPositive :: Int -> Int
keepPositive x = if x < 0 then 0 else x ❶
```

❶ We're forcing `x` to be `0` if it's negative otherwise we leave `x` alone.

If we didn't have the `else` portion then what would `keepPositive x` evaluate to when `x` is `-1`?

In Javascript, there is a similar `if` expression that requires the else part:

```
const keepPositive = x => x < 0 ? 0 : x
```

Javascript also has an `if` statement, but in PureScript everything is an expression and there is no way to evaluate something in one case and nothing in the other. We are always evaluating an expression or calling a Function and then doing something with that result.

If you have an `if` statement with no `else` then you're typically doing one of a few things, performing a calculation or calling a Function and storing its results into a variable, or you're calling a Function for its side-effects, etc.

First, there are no side-effects in Pure Functional Programming so that case is out for us.

Second, calling a Function for its return Value or doing a calculation for its results is the same thing. And we can do that too, but we cannot OPTIONALLY assign that result to a variable because that assumes one of two things. Either the variable already has a Value which means we'd be mutating it or it has no Value and therefore is NULL.

Both of these scenarios are forbidden in PureScript. This is why we only have an `if` expression.

3.3.2. Case Expression

We can do logical branching using a `case` expression but it's limited to Pattern Matches:

```
data ContactMethod
  = Phone
  | Email
  | Fax

keepModern :: ContactMethod -> ContactMethod
keepModern preferredContactMethod =
  case preferredContactMethod of
    Phone -> Phone
    Email -> Email
    Fax -> Email ❶
```

❶ If the preferred contact method is `Fax` we use `Email` instead.

Technically, we could rewrite `keepPositive` using `case`:

```
keepPositive :: Int -> Int
keepPositive x = case x < 0 of
  true -> 0
  false -> x
```

While this technically is valid, this version is clunky compared to the `if-then-else` one.

3.3.3. Pattern Matching

Pattern Matching is very similar to using `case`:

```
data ContactMethod
  = Phone
  | Email
  | Fax

keepModern :: ContactMethod -> ContactMethod ❶
keepModern Phone = Phone
keepModern Email = Email
keepModern Fax = Email
```

❶ This version is slightly easier to read than the `case` expression version.

There is no way to write `keepPositive` with ONLY Pattern Matching. We need a way to have an `if-like` operation, which Pattern Matching doesn't allow.

If only there was another way…

3.3.4. Guards

Guards are a way to specify `if` logic in a concise and readable manner. The syntax is a bit strange when first encountered:

```
keepPositive :: Int -> Int
keepPositive x ❶
  | x < 0     = 0 ❷ ❸
  | otherwise = x ❷ ❹
```

❶ Notice NO equal sign when using Guards. This fact is really easy to forget.

❷ The vertical bar, `|`, is the Guard. What follows is a Boolean expression, an equal sign and the Value of the expression if the Boolean expression is `true`.

❸ This line says if `x < 0` is `true` then return `0`.

❹ `otherwise` is defined to equal `true` and only exists to make reading Guards nicer. It is the catchall case where we simply return `x`.

We can also use Guards in `case` expresssions:

```
data ContactMethod
  = Phone
  | Email
  | Fax

keepModernIfYoung :: Int -> ContactMethod -> ContactMethod
keepModernIfYoung age preferredContactMethod =
  case preferredContactMethod of
    Phone -> Phone
    Email -> Email
    Fax | age < 40  -> Email ❶ ❸
        | otherwise -> Fax ❷ ❸
```

❶ We first Pattern Match on Fax and then the Guards kick in. If `age` is younger than `40`, we use `Email`.

❷ Our catchall case will be for anyone `40` and older, and we'll use `Fax`.

❸ Don't forget to use `->` instead of `=` for Guards in `case` expressions. This too is easy to forget.

Here's another example of Guards in a `case`:

```
noBiggerThan10 :: Maybe Int -> Int
noBiggerThan10 x = case x of
  Just x   | x > 10    -> 10 ❶
           | otherwise -> x
  Nothing              -> 0
```

❶ The Pattern Match is on the left of `|`, then the conditional, followed by the arrow, and finally the Value. Notice how the `x` in the pattern `Just x` can be used on the right side of the Guard.

Guards and Pattern Matching can work together:

```
takeWhile :: ∀ a. (a -> Boolean) -> List a -> List a ❶
takeWhile p (x : xs) | p x = x : takeWhile p xs ❷
takeWhile _ _ = Nil ❸
```

❶ Takes a Predicate and a `List`, and takes from the head of the `List` while the Predicate is met. It stops as soon as the Predicate returns `false` and returns all the elements it took up to that point in a `List`.

❷ `p x` checks the head of the `List`, `x`, against the Predicate, `p`. If `true` it adds `x` to the head of the recursive call to `takeWhile` with the tail. If, however, it's `false`, there is no catchall Guard and so it goes to the other version of `takeWhile`

❸ This version covers 2 cases. First, it's the `false` case for the first version with the Guard. Second, it's the case for an empty `List`, since `Nil` will not match the second Parameter of the first version of takeWhile, viz. (x : xs).

Let’s review what the pattern x : xs will match:

```
1 : 2 : 3 : Nil   (x = 1, xs = 2 : 3 : Nil)
1 : 2 : Nil       (x = 1, xs = 2 : Nil)
1 : Nil           (x = 1, xs = Nil)
Nil               (NO MATCH)
```

A ***Predicate*** is a Function that takes a Value and returns a Boolean based on some condition regarding that Value, e.g. `isOdd :: Int -> Boolean` will take an `Int` and return `true` if it’s odd.

3.4. Lambda Functions

Earlier, when we were learning a little bit about Lambda Calculus, we saw PureScript’s equivalent to Functions in Lambda Calculus:

```
\x -> x + 1
```

This Function is an Anonymous Function, i.e. has no name. We could name it by calling it f:

```
f :: Int -> Int
f = \x -> x + 1
```

Since we’ve named this f, there’s no reason to use the Lambda. We could just as easily write f with the Parameter on the other side of the equal sign:

```
f :: Int -> Int
f x = x + 1
```

So why is this syntax supported. What’s the real benefit of a Lambda?

This syntax is helpful when you need to pass a Function to another Function, but it’s not worthy of a proper name since it’ll only be used once:

```
filter (\x -> x < 10) [1, 2, 3, 10, 20, 30] -- [1, 2, 3]
```

Here the Lambda Function is a Predicate that checks to see if the Value is less than 10.

`filter` will keep anything from the `Array` that passes the test. It'll iterate through our `Array` and call our Lambda passing the element from the `Array` and if the Lambda returns `true`, then `filter` will keep it, otherwise it's filtered out.

Remember there are NO mutations, therefore the `Array` that you get back from `filter` will be a new `Array`.

To specify multiple Parameters in a Lambda is very similar to normal Functions:

```
\x y -> x + y
```

which is syntactical sugar for:

```
\x -> \y -> x + y
```

If you're used to Javascript, you've seen a similar pattern for manually writing curried Functions:

```
x => y => x + y // Javascript
```

There are times when Functions have more Parameters than is specified in the Type Signatures. This can be really confusing when you first encounter this, but taking into consideration Lambdas can help.

To illustrate this, let's examine the `compose` Function, usually called via its Binary Operator, `<<<`, which composes two Functions:

```
compose :: ∀ a b c. (b -> c) -> (a -> b) -> (a -> c) ❶
compose f g x = f (g x) ❷ ❸
```

❶ `compose` takes 2 Parameters, both Functions, and returns a Function.

❷ We've specified 3 parameters here. What's up?

❸ `f (g x)` is of Type `c` which does NOT match the return Value in the Type Signature. What's going on?

It turns out that this Function is equivalent to writing it with a Lambda:

```
compose :: ∀ a b c. (b -> c) -> (a -> b) -> (a -> c)
compose f g = \x -> f (g x) ❶ ❷
```

❶ Now it takes ONLY 2 Parameters.

❷ Now it returns a Function from `a` to `c`. The Lambda on the right side of the equal sign takes an `x` of Type `a` and returns `f (g x)` of Type `c`.

We can freely move Parameters across the equals sign as long as we move the rightmost Parameter first and maintain the correct order:

```
f1 x y z =           x + y + z
f2 x y   = \z ->     x + y + z ❶
f3 x     = \y z ->   x + y + z ❷
f4       = \x y z -> x + y + z ❸
```

❶ First we move the rightmost Parameter `z`.

❷ Then we move the next rightmost Parameter `y` and maintain the proper order in the Lambda, i.e. `y` comes before `z`, just like it did in `f1`.

❸ And finally we move the `x` maintaining its proper position.

To better understand the maintainence of Parameter order, let's look at `f3`. It takes `x` and will return a Function that takes `y` first and then `z`. This maintains the order `x`, `y`, `z` found in `f1`.

In fact, all the Functions are equivalent:

```
f1 = f2 = f3 = f4
```

We've answered the question about the number of Parameters but let's return to our original conumdrum to see another way to look at this:

```
compose :: ∀ a b c. (b -> c) -> (a -> b) -> (a -> c) ❶
compose f g x = f (g x) ❶
```

❶ `f (g x)` is of Type `c` which is NOT what the return Value of this Function shows.

Remember that Type Signatures are Right-Associative which means they have implied Parentheses on the right. This means that the Parentheses on `(a -> c)` are redundant and we can remove them:

```
compose :: ∀ a b c. (b -> c) -> (a -> b) -> a -> c ❶
compose f g x = f (g x) ❷
```

❶ Now we have 3 Parameters!

❷ `f (g x)` is of Type `c` which is our return Type.

Both ways of looking at the Type Signature are valid as are both ways of understanding our original dilemma.

3.5. Wildcards

So far, we've seen `_` used as a "don't-care" variable.

There are other times when it's used as a ***Wildcard***.

3.5.1. Case Expression

```
data ContactMethod
  = Phone
  | Email
  | Fax

-- verbose version

keepModernIfYoung :: Int -> ContactMethod -> ContactMethod
keepModernIfYoung age preferredContactMethod =
  case preferredContactMethod of ❶
    Phone -> Phone
    Email -> Email
    Fax | age < 40  -> Email
        | otherwise -> Fax

-- simpler with a wildcard

keepModernIfYoung' :: Int -> ContactMethod -> ContactMethod
keepModernIfYoung' age = case _ of ❷
  Phone -> Phone
  Email -> Email
  Fax | age < 40  -> Email
      | otherwise -> Fax
```

❶ Notice that we had to provide a name for a Parameter that would be immediately used in a `case` expression and never again.

❷ Here we write the Function in Point-free notation. This allows the `_` to stand for the second Parameter. It's still a "don't-care" Variable since we didn't name it, yet as a Wildcard, we actually use the Value.

The following 2 lines of code are equivalent:

```
keepModernIfYoung' age = case _ of

keepModernIfYoung' age = \someNameIHaveToThinkOf ->
  case someNameIHaveToThinkOf of
```

I think this illustrates well the benefits of using Wildcards.

3.5.2. Operator Sections

Wildcards can also be used in ***Operator Sections***:

```
filter (\x -> x < 10) [1,2,3,10,20,30] -- [1,2,3] ❶
filter (_ < 10) [1,2,3,10,20,30] -- [1,2,3] ❷
filter (10 <= _) [1,2,3,10,20,30] -- [10,20,30] ❸
```

❶ This is our filter Function call from above that uses a Lambda.

❷ This is using an Operator Section with the Wildcard as the left side of the Operator. This is equivalent to the previous Function call.

❸ This is using the Wildcard on the other side of the Operator.

Operator Sections are Binary Operators with one or two Wildcard variables set off by Parentheses.

The following are also valid Operator Sections:

```
(_ <> "suffix")    -- append the string "suffix" ❶
(_ / 10)           -- div by 10 ❶
(10 / _)           -- div into 10 ❶
(_ == _)           -- equal check ❷
```

❶ These Functions take 1 Parameter.

❷ This Function takes 2 Parameters.

3.5.3. Records

Wildcards can be using in Records:

```
\name age -> {name: name, age: age} ❶
{ name: _, age: _ } ❷
```

❶ A Lambda that takes 2 Parameters and returns a record populated with the values from those Parameters.

❷ This is the Wildcard equivalent of the previous line. It's very important that the order of Parameters is maintained, i.e. the first Wildcard encountered from Left to Right is the first Parameter, the second Wildcard is the second Parameter and so on.

Record updates can also have Wildcards:

```
\name age -> person { name = name, age = age } ❶
person { name = _, age = _} ❷
```

❶ Here's a Lambda that takes `person` and updates 2 of its fields (`person` could have more than just 2 fields) and returns a new `Record` with just those 2 fields modified. Remember, there are no mutations, so `person` is unchanged.

❷ The same can be accomplished with Wildcards using this syntax.

Another Wildcard example for updates:

```
\person -> person { age = 18 } ❶
_ { age = 18 } ❷
```

❶ The Lambda takes a `person` and updates the `age` returning a new `Record`.

❷ The Wildcard equivalent. This syntax looks odd at first.

3.6. Bindings

We have 2 ways to bind Values to variables. (Don't forget, Functions are also Values in Functional Programming.)

3.6.1. Where

The keyword `where` allows us to define things after the fact in a Function definition:

```
multSum :: Int -> Int -> Tuple Int Int ❶
multSum x y = Tuple mult sum ❷
  where
    mult = x * y ❸ ❺
    sum = x + y ❹ ❺
```

❶ `multSum` takes 2 `Ints` and returns a `Tuple` of `Ints`.

❷ Variables `mult` and `sum` are used here but have not been defined yet.

❸ `mult` is bound to `x * y`.

❹ `sum` is bound to `x + y`.

❺ The `where` section ONLY has access to variables at the same level as the Function, `multSum`, i.e its Parameters. Any local Function variables are NOT accessible to `where`.

The `where` section is private to the Function `multSum`.

This style of binding doesn't bog down the human reader with too many upfront details. The terms `mult` and `sum` are introduced to the reader first and then their actual definitions are an afterthought in the `where` section.

Sometimes, it's best to defer the gory details of your computation and first present the high-level concepts in the code.

The `Tuple mult sum` tells us a lot. It says we have a multiplication of some sort in the first position of the `Tuple` and an addition in the second.

Once we get to the `where` we can see the details of those calculations. And sometimes, we never need to delve this far, depending on the Function.

For line continuations, the next line has to be indented PAST the first character of variable name:

```
where
  mult = x * y
  sum = x
    + y ❶
```

❶ The first character has to be indented at least one space past the `s` in `sum`.

3.6.2. Let

In constrast, `let` defines things up front:

```
multSum' :: Int -> Int -> Tuple Int Int
multSum' x y =
  let mult = x * y ❶
      sum = x + y in ❷
  Tuple mult sum
```

❶ A `let`, like `where`, can have multiple definitions. With `let`, we are burdening our reader with details here first and then specifying the overall computation.

❷ The `let` block is terminated with the reserved word `in`.

The bindings in the `let` block are only valid within the `in` block.

Unlike where which must be part of a NAMED Function, let can be used as part of ANY expression:

```
filter (\n -> let n2 = n * n in n == n2) [0, 1, 2] -- [0, 1]
```

If you try this with a where, you'll get a compiler error:

```
-- ILLEGAL
filter (\n -> n == n2 where n2 = n * n) [0, 1, 2] -- COMPILER ERROR!!
```

For line continuations, the indentation is similar to where:

```
let mult = x * y
    sum = x
      + y in
Tuple mult sum
```

3.7. Binary Operators

We've seen Binary Operators in action but we failed to define them.

A ***Binary Operator*** is an Operator represented by a Symbol that is a Function of 2 Parameters. The Value on the lefthand side of the Operator is the first Parameter and the Value on the right is the second.

Let's look at an example:

```
data List a = Nil | Cons a (List a)

append :: ∀ a. List a -> List a -> List a ❶
append xs Nil = xs ❷
append Nil ys = ys ❸
append (Cons x xs) ys = Cons x (append xs ys) ❹

infixr 5 append as <> ❺
```

❶ append will append the second List to the first.

❷ The case where the second List is empty.

❸ The case where the first List is empty.

❹ The general case where we remove the head of the first List and PREPEND it to the append of the tail of the first `List` with the whole second `List`.

❺ Defining the operator `<>` as the infixed, Right-Associative operator with Precedence 5.

3.7.1. Associativity

There are 3 types of Associativity for Binary Operators:

- `infixr` = Right-Associative
- `infixl` = Left-Associative
- `infix` = None

The operator `<>` is Right-Associative, which means the following two lines are equivalent:

```
l1 <> l2 <> l3 <> l4
(l1 <> (l2 <> (l3 <> l4)))
```

Notice how the Parentheses collect on the Right.

The operator `+` is Left-Associative, which means the following two lines are equivalent:

```
n1 + n2 + n3 + n4
(((n1 + n2) + n3) + n4)
```

Notice how the Parentheses collect on the Left.

The operator `==` is Non-Associative, i.e. is defined using `infix`. This means that repeated use of the operator (or use with other Non-Associative operators) in an expression MUST have EXPLICIT Parentheses:

```
true == false == false     -- COMPILER ERROR!!
(true == false) == false   -- compiles with Value of true
true == (1 == 42)          -- compiles with Value of false
```

3.7.2. Precedence

The `*` operator has a Precedence of 7, whereas `+` has a Precedence of 6, which means the following two lines are equivalent:

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

The higher the Precedence number, the sooner it's evaluated. Here `*` is higher than `+` and so it's evaluated first.

In our `append` definition above, we specified:

```
infixr 5 append as <>
```

That defines the operator `<>` as the equivalent to calling `append` infixed with Right Associativity and a Precedence of 5. That means it's evaluated before any other operators that are part of the same expression that have a lower Precedence, i.e. 4, 3, 2, 1, and 0.

3.7.3. Fixity

`infix` means Infixed Operator, i.e. it comes between its two values:

```
l1 <> l2
```

Compare that with the Prefixed equivalent:

```
append l1 l2
```

We can use an Infixed Operator in Prefixed Position and we can use a Function in Infixed Position:

```
(<>) l1 l2 ❶
l1 `append` l2 ❷
```

❶ The Operator must be surrounded by Parentheses to be used in Prefixed Position.

❷ The Function name must be surrounded by Back-ticks to be used in Infixed Position.

Infixing a named Function makes sense when it helps readability, e.g. `elem` is a Predicate that returns `true` if a Value is an element of an `Array`:

```
if 42 `elem` [1, -5, 42] then "YES" else "NO"
```

We can read this as: `if 42 is an element of [1, -5, 42]…`

3.8. Comments

We've seen inline comments up to this point, e.g.:

```
add :: Int -> Int -> Int
add x y = x + y -- add 2 numbers
```

The inline comment is prefixed with `--`.

Block comments can be accomplished as follows:

```
{-  This Function will add two numbers
    and return the answer, which is
    the addition of those two numbers.
-}
add :: Int -> Int -> Int
add x y = x + y -- add 2 numbers
```

Comments that start with a pipe character, `|`, are considered Documentation for tools such as `psc-docs` and `Pursuit`, the online repository of PureScript API documentation. Pursuit can be found at https://pursuit.purescript.org/.

3.9. Inferring Functionality from Type Signatures

All Functions have Type Signatures whether defined explicitly by the developer or determined implicitly by the compiler based on how the developer uses the Parameters.

Best practices deems that at least all top-level Functions in a module have explicit Type Signatures.

Not only does this help by improving the readability of the codebase, but it helps the compiler to produce better error messages. But Type Signatures aren’t just helpful to the compiler. We can imply a lot from just a Type Signature.

Take the following Type Signature:

```
f :: Int -> Int -> Int
```

Take a guess at what kind of Function `f` might be.

Here is a list of possible Functions for `f`:

```
max :: Int -> Int -> Int
min :: Int -> Int -> Int
add :: Int -> Int -> Int
sub :: Int -> Int -> Int
mult :: Int -> Int -> Int
div :: Int -> Int -> Int
```

And we could go on from here.

But what about more abstract Type Signatures such as:

```
f :: ∀ a. a -> a
```

How many Functions can you write that perform this? Try to list them all out before reading any further.

Remember, that `a` is an unknown Type to the developer of `f`. It can be ANY Type, e.g. an `Int`, `String`, `Array Unit`, `List (Maybe Int)`, anything.

Now how many Functions can you think of?

The way to think this through is to realize that you're given a value of some UNKNOWN Type `a` and you have to return a value of some UNKNOWN Type `a`. You cannot write any code to look at the `a` because you don't know what `a` is. But you still need to produce an `a`. And the only `a` you have is the one you've been given.

Therefore, there is only 1 Function that can have this signature:

```
identity :: ∀ a. a -> a
identity x = x
```

`identity` takes the only `a` it has and returns it back unchanged since it doesn't know anything about `a`.

What about the following Function:

```
f :: ∀ a b. a -> b
```

How many Functions can your write with this Type Signature?

Thinking this through, we are given an `a` and must produce a `b`. But we don't know what Type `a` is and we don't know what Type `b` is. So how are we supposed to convert an UNKNOWN `a` into an UNKNOWN `b`?

The answer is we cannot. There are no Functions with this Type Signature. If you find this hard to believe, try to imagine writing a Function where `a` is an `Int` and `b` is a `String`. But that SAME Function must also work when `a` is a `String` and `b` is an `Int`. Remember, the caller of your Function determines which Concrete Types `a` and `b` are, not you.

How about this Type Signature:

```
f :: ∀ a. List a -> Maybe a
```

Try to enumerate how many Functions you could write before moving on.

In the above example, we have a `List` of elements which are of Type `a`, i.e. UNKNOWN Type to us. And we must produce an `a` and since our only source of `a’s` is the `List`, it must come from the `List`. We also have the possibility of failure as signified by the `Maybe`.

Here’s a reasonable list of Functions with that Type Signature:

```
first :: ∀ a. List a -> Maybe a ❶
last :: ∀ a. List a -> Maybe a ❷
constIndex :: ∀ a. List a -> Maybe a ❸
```

❶ Tries to return the first thing in the `List`. Returns `Nothing` if the `List` is empty.

❷ Tries to return the last thing in the `List` or `Nothing` if it fails, i.e. for an empty `List`.

❸ Tries to return the element at some hardcoded index. Returns `Nothing` if the index is out of bounds.

It turns out that `first` and `last` are just special cases of `constIndex`, so for all practical purposes, there is only 1 unique Functions with this Type Signature.

What about:

```
f :: ∀ a. List a -> a
```

What does this Function appear to do?

Given a `List`, this Function ALWAYS produces an `a`. The problem is that this promise cannot be honored if we pass it an empty `List`.

There are no ***Total Functions*** that can be written with this Type Signature. We can, however, write a ***Partial Function***, but it would crash on the empty `List` case.

A Partial Function is one where all of the cases are not handled. A Total Function is one where all possible calling scenarios have been accounted for. PureScript discourages Partial Functions.

How can we take this Partial Function and make it Total?

Here's the Type Signature for a Total version:

```
f :: ∀ a. a -> List a -> a
```

Can you see what this does? Try to imagine the Function that we'd write for this Type Signature before moving on.

The newly added Parameter is a Default Value:

```
alwaysFirst :: ∀ a. a -> List a -> a
alwaysFirst def list = ... ❶
alwaysLast :: ∀ a. a -> List a -> a
alwaysLast def list = ... ❶
```

❶ The details of the Function are not important here.

If given an empty `List`, these Functions will return the supplied default.

Let's try another Type Signature:

```
f :: ∀ a. a -> Int -> List a -> a
```

This is pretty close to the previous Type Signature with the addition of an `Int` Parameter. What can we do with that Parameter?

The most logical explanation is that the `Int` is an index into the `List`:

```
getAt :: ∀ a. a -> Int -> List a -> a
getAt def index list = ... ❶
```

❶ The details of the Function are not important here.

Once again, if given an empty `List`, this Function will return the supplied default.

What would you put in place of the `???` in the following:

```
f :: ∀ a. Int -> List a -> ???
```

This looks like our `getAt` Function above, but it's just missing the default Parameter. That means that we could fail in the case of an empty `List`.

That means we have:

```
getAt' :: ∀ a. Int -> List a -> Maybe a ❶
getAt` index list = ... ❷
```

❶ `???` was replaced by `Maybe a` to account for the possibility of a failure.

❷ The details of the Function are not important here.

Here's another one:

```
f :: ∀ a b. a -> Int -> (a -> b) -> List a -> b
```

This may seem pretty complex, but it's actually not. It looks a lot like `getAt` but with the added Type `b` and the extra Parameter `a -> b`.

So this Function simply does what `getAt` does, but before it returns a Value, it converts it from Type `a` to `b` using the supplied conversion Function (3rd Parameter).

Here's what this Function looks like:

```
getAtConvert :: ∀ a b. a -> Int -> (a -> b) -> List a -> b
getAtConvert def index convert list = ... ❶
```

❶ The details of the Function are not important here.

But wait. I thought we couldn't have a Function from `a -> b`. What gives?

While it's true that we cannot write a Function from ANY `a` to ANY `b`, we could write a Function from `Int` to `String` and then call `getAtConvert` with it:

```
intToString :: Int -> String
intToString i = show i

getAtConvert (-1) 2 intToString (1 : 2 : 3 : Nil) ❶
```

❶ Here our default value is `-1` and our index is `2`.

When we call this Function with `intToString`, our Type Parameter `a` gets unified with `Int` and `b` gets unified with `String`. Our `convert` Function's Type is `a -> b` or, in this particular case, `Int -> String`. A Function from `Int` to `String` is a Function we can write.

We cannot have a free-standing Function with the Type Signature of `a -> b` since we cannot write code for converting ANY `a` to ANY `b`. But in this case, the caller of our Function can pick Concrete Types for `a` and `b`, write a conversion Function for those Concrete Types and then pass that Function to us.

We don't have any idea what `a` and `b` will be when we write our Function, so the best we can do is take something of Type `a` and give it to that Function to get a Value of Type `b`.

Notice that we didn't need to look at the implementation of the Function to guess what that Function does and what roles the Parameters played in the Function.

For example, if we have the following:

```
inBetween :: Int -> Int -> Boolean
```

We have enough information to make a pretty good guess as to what it does and what each Parameter SHOULD be. In this example, convention would dictate that we have a starting and ending number in that order. The only real question is whether `inBetween` is inclusive of its endpoints or not.

If we see:

```
singleton :: ∀ a. a -> List a
```

We can surmize that this takes a Value of Type `a` and puts it into a `List`.

When we see:

```
f :: ∀ a. a -> a -> Maybe a
```

The best we can surmize here is that `f` takes 2 values of `a` and possibly returns one of them based on some internal criteria.

And with:

```
f :: ∀ a. a -> (a -> Boolean) -> Maybe a
```

We can see that we're passing the criteria (Predicate) for returning `a`. The criteria is the 2nd Parameter which will take an `a` and return `true` or `false`, which we can reasonably assume will be returned if `true` and not if `false`.

A lot can be determined by just looking at the Type Signatures of Functions. This process will come it handy when we start coding and we're looking at library documentation that's somewhat vague or incomplete, which unfortunately is more the rule than the exception.

3.10. Summary

We’ve learned a lot in this Chapter about some of the basic parts of PureScript from Types to common language constructs. There will be more to come. And while theory is important, getting some hands on experience can help cement that theoretical understanding. And often times, we think we understand something only to learn otherwise when we attempt to use our newfound knowledge.

You will probably find yourself returning to this Chapter as you work out the coming exercises. Now it’s time to get our hands dirty.