# Summary

[Foreword](https://github.com/purescript-contrib/purescript-book/blob/master/README.md)

* [Introduction](https://github.com/purescript-contrib/purescript-book/blob/master/text/chapter1.md)
* [Getting Started](https://github.com/purescript-contrib/purescript-book/blob/master/text/chapter2.md)
* [Functions and Records](https://github.com/purescript-contrib/purescript-book/blob/master/text/chapter3.md)
* [Recursion, Maps And Folds](https://github.com/purescript-contrib/purescript-book/blob/master/text/chapter4.md)
* [Pattern Matching](https://github.com/purescript-contrib/purescript-book/blob/master/text/chapter5.md)
* [Type Classes](https://github.com/purescript-contrib/purescript-book/blob/master/text/chapter6.md)
* [Applicative Validation](https://github.com/purescript-contrib/purescript-book/blob/master/text/chapter7.md)
* [The Effect Monad](https://github.com/purescript-contrib/purescript-book/blob/master/text/chapter8.md)
* [Asynchronous Effects](https://github.com/purescript-contrib/purescript-book/blob/master/text/chapter9.md)
* [The Foreign Function Interface](https://github.com/purescript-contrib/purescript-book/blob/master/text/chapter10.md)
* [Monadic Adventures](https://github.com/purescript-contrib/purescript-book/blob/master/text/chapter11.md)
* [Canvas Graphics](https://github.com/purescript-contrib/purescript-book/blob/master/text/chapter12.md)
* [Generative Testing](https://github.com/purescript-contrib/purescript-book/blob/master/text/chapter13.md)
* [Domain-Specific Languages](https://github.com/purescript-contrib/purescript-book/blob/master/text/chapter14.md)

# Introduction

## Functional JavaScript

Functional programming techniques have been making appearances in JavaScript for some time now:

* Libraries such as [UnderscoreJS](https://underscorejs.org) allow the developer to leverage tried-and-trusted functions such as map, filter and reduce to create larger programs from smaller programs by composition:
* var sumOfPrimes =
* \_.chain(\_.range(1000))
* .filter(isPrime)
* .reduce(function(x, y) {
* return x + y;
* })

.value();

* Asynchronous programming in NodeJS leans heavily on functions as first-class values to define callbacks.
* import { readFile, writeFile } from 'fs'
* readFile(sourceFile, function (error, data) {
* if (!error) {
* writeFile(destFile, data, function (error) {
* if (!error) {
* console.log("File copied");
* }
* });
* }

});

* Libraries such as [React](https://reactjs.org) and [virtual-dom](https://github.com/Matt-Esch/virtual-dom) model views as pure functions of application state.

Functions enable a simple form of abstraction which can yield great productivity gains. However, functional programming in JavaScript has its own disadvantages: JavaScript is verbose, untyped, and lacks powerful forms of abstraction. Unrestricted JavaScript code also makes equational reasoning very difficult.

PureScript is a programming language which aims to address these issues. It features lightweight syntax, which allows us to write very expressive code which is still clear and readable. It uses a rich type system to support powerful abstractions. It also generates fast, understandable code, which is important when interoperating with JavaScript, or other languages which compile to JavaScript. All in all, I hope to convince you that PureScript strikes a very practical balance between the theoretical power of purely functional programming, and the fast-and-loose programming style of JavaScript.

## Types and Type Inference

The debate over statically typed languages versus dynamically typed languages is well-documented. PureScript is a statically typed language, meaning that a correct program can be given a type by the compiler which indicates its behavior. Conversely, programs which cannot be given a type are incorrect programs, and will be rejected by the compiler. In PureScript, unlike in dynamically typed languages, types exist only at compile-time, and have no representation at runtime.

It is important to note that in many ways, the types in PureScript are unlike the types that you might have seen in other languages like Java or C#. While they serve the same purpose at a high level, the types in PureScript are inspired by languages like ML and Haskell. PureScript's types are expressive, allowing the developer to assert strong claims about their programs. Most importantly, PureScript's type system supports type inference - it requires far fewer explicit type annotations than other languages, making the type system a tool rather than a hindrance. As a simple example, the following code defines a number, but there is no mention of the Number type anywhere in the code:

iAmANumber =

let square x = x \* x

in square 42.0

A more involved example shows that type-correctness can be confirmed without type annotations, even when there exist types which are unknown to the compiler:

iterate f 0 x = x

iterate f n x = iterate f (n - 1) (f x)

Here, the type of x is unknown, but the compiler can still verify that iterate obeys the rules of the type system, no matter what type x might have.

In this book, I will try to convince you (or reaffirm your belief) that static types are not only a means of gaining confidence in the correctness of your programs, but also an aid to development in their own right. Refactoring a large body of code in JavaScript can be difficult when using any but the simplest of abstractions, but an expressive type system together with a type checker can even make refactoring into an enjoyable, interactive experience.

In addition, the safety net provided by a type system enables more advanced forms of abstraction. In fact, PureScript provides a powerful form of abstraction which is fundamentally type-driven: type classes, made popular in the functional programming language Haskell.

## Polyglot Web Programming

Functional programming has its success stories - applications where it has been particularly successful: data analysis, parsing, compiler implementation, generic programming, parallelism, to name a few.

It would be possible to practice end-to-end application development in a functional language like PureScript. PureScript provides the ability to import existing JavaScript code, by providing types for its values and functions, and then to use those functions in regular PureScript code. We'll see this approach later in the book.

However, one of PureScript's strengths is its interoperability with other languages which target JavaScript. Another approach would be to use PureScript for a subset of your application's development, and to use one or more other languages to write the rest of the JavaScript.

Here are some examples:

* Core logic written in PureScript, with the user interface written in JavaScript.
* Application written in JavaScript or another compile-to-JS language, with tests written in PureScript.
* PureScript used to automate user interface tests for an existing application.

In this book, we'll focus on solving small problems with PureScript. The solutions could be integrated into a larger application, but we will also look at how to call PureScript code from JavaScript, and vice versa.

## Prerequisites

The software requirements for this book are minimal: the first chapter will guide you through setting up a development environment from scratch, and the tools we will use are available in the standard repositories of most modern operating systems.

The PureScript compiler itself can be downloaded as a binary distribution, or built from source on any system running an up-to-date installation of the GHC Haskell compiler, and we will walk through this process in the next chapter.

The code in this version of the book is compatible with versions 0.15.\* of the PureScript compiler.

## About You

I will assume that you are familiar with the basics of JavaScript. Any prior familiarity with common tools from the JavaScript ecosystem, such as NPM and Gulp, will be beneficial if you wish to customize the standard setup to your own needs, but such knowledge is not necessary.

No prior knowledge of functional programming is required, but it certainly won't hurt. New ideas will be accompanied by practical examples, so you should be able to form an intuition for the concepts from functional programming that we will use.

Readers who are familiar with the Haskell programming language will recognize a lot of the ideas and syntax presented in this book, because PureScript is heavily influenced by Haskell. However, those readers should understand that there are a number of important differences between PureScript and Haskell. It is not necessarily always appropriate to try to apply ideas from one language in the other, although many of the concepts presented here will have some interpretation in Haskell.

## How to Read This Book

The chapters in this book are largely self contained. A beginner with little functional programming experience would be well-advised, however, to work through the chapters in order. The first few chapters lay the groundwork required to understand the material later on in the book. A reader who is comfortable with the ideas of functional programming (especially one with experience in a strongly-typed language like ML or Haskell) will probably be able to gain a general understanding of the code in the later chapters of the book without reading the preceding chapters.

Each chapter will focus on a single practical example, providing the motivation for any new ideas introduced. Code for each chapter are available from the book's [GitHub repository](https://github.com/purescript-contrib/purescript-book). Some chapters will include code snippets taken from the chapter's source code, but for a full understanding, you should read the source code from the repository alongside the material from the book. Longer sections will contain shorter snippets which you can execute in the interactive mode PSCi to test your understanding.

Code samples will appear in a monospaced font, as follows:

module Example where

import Effect.Console (log)

main = log "Hello, World!"

Commands which should be typed at the command line will be preceded by a dollar symbol:

$ spago build

Usually, these commands will be tailored to Linux/Mac OS users, so Windows users may need to make small changes such as modifying the file separator, or replacing shell built-ins with their Windows equivalents.

Commands which should be typed at the PSCi interactive mode prompt will be preceded by an angle bracket:

> 1 + 2

3

Each chapter will contain exercises, labelled with their difficulty level. It is strongly recommended that you attempt the exercises in each chapter to fully understand the material.

This book aims to provide an introduction to the PureScript language for beginners, but it is not the sort of book that provides a list of template solutions to problems. For beginners, this book should be a fun challenge, and you will get the most benefit if you read the material, attempt the exercises, and most importantly of all, try to write some code of your own.

## Getting Help

If you get stuck at any point, there are a number of resources available online for learning PureScript:

* The [PureScript Discord server](https://discord.gg/vKn9up84bp) is a great place to chat about issues you may be having. The server is dedicated to chat about PureScript
* The [Purescript Discourse Forum](https://discourse.purescript.org/) is another good place to search for solutions to common problems. Questions you ask here will be available to help future readers, whereas on Slack, message history is only kept for approximately 2 weeks.
* [PureScript: Jordan's Reference](https://github.com/jordanmartinez/purescript-jordans-reference) is an alternative learning resource that goes into great depth. If a concept in this book is difficult to understand, consider reading the corresponding section in that reference.
* [Pursuit](https://pursuit.purescript.org) is a searchable database of PureScript types and functions. Read Pursuit's help page to [learn what kinds of searches you can do](https://pursuit.purescript.org/help/users).
* The unofficial [PureScript Cookbook](https://github.com/JordanMartinez/purescript-cookbook) provides answers via code to "How do I do X?"-type questions.
* The [PureScript documentation repository](https://github.com/purescript/documentation) collects articles and examples on a wide variety of topics, written by PureScript developers and users.
* The [PureScript website](https://www.purescript.org) contains links to several learning resources, including code samples, videos and other resources for beginners.
* [Try PureScript!](https://try.purescript.org) is a website which allows users to compile PureScript code in the web browser, and contains several simple examples of code.

If you prefer to learn by reading examples, the purescript, purescript-node and purescript-contrib GitHub organizations contain plenty of examples of PureScript code.

## About the Author

I am the original developer of the PureScript compiler. I'm based in Los Angeles, California, and started programming at an early age in BASIC on an 8-bit personal computer, the Amstrad CPC. Since then I have worked professionally in a variety of programming languages (including Java, Scala, C#, F#, Haskell and PureScript).

Not long into my professional career, I began to appreciate functional programming and its connections with mathematics, and enjoyed learning functional concepts using the Haskell programming language.

I started working on the PureScript compiler in response to my experience with JavaScript. I found myself using functional programming techniques that I had picked up in languages like Haskell, but wanted a more principled environment in which to apply them. Solutions at the time included various attempts to compile Haskell to JavaScript while preserving its semantics (Fay, Haste, GHCJS), but I was interested to see how successful I could be by approaching the problem from the other side - attempting to keep the semantics of JavaScript, while enjoying the syntax and type system of a language like Haskell.

I maintain [a blog](https://blog.functorial.com), and can be [reached on Twitter](https://twitter.com/paf31).

## Acknowledgements

I would like to thank the many contributors who helped PureScript to reach its current state. Without the huge collective effort which has been made on the compiler, tools, libraries, documentation and tests, the project would certainly have failed.

The PureScript logo which appears on the cover of this book was created by Gareth Hughes, and is gratefully reused here under the terms of the [Creative Commons Attribution 4.0 license](https://creativecommons.org/licenses/by/4.0/).

Finally, I would like to thank everyone who has given me feedback and corrections on the contents of this book.

# Getting Started

## Chapter Goals

In this chapter, we'll set up a working PureScript development environment, solve some exercises, and use the tests provided with this book to check our answers. You may also find a [video walkthrough of this chapter](https://www.youtube.com/watch?v=GPjPwb6d-70) helpful if that better suits your learning style.

## Environment Setup

First, work through this [Getting Started Guide](https://github.com/purescript/documentation/blob/master/guides/Getting-Started.md) in the Documentation Repo to setup your environment and learn a few basics about the language. Don't worry if the code in the example solution to the [Project Euler](http://projecteuler.net/problem=1) problem is confusing or contains unfamiliar syntax. We'll cover all of this in great detail in the upcoming chapters.

## Solving Exercises

Now that you've installed the necessary development tools, clone this book's repo.

git clone https://github.com/purescript-contrib/purescript-book.git

The book repo contains PureScript example code and unit tests for the exercises that accompany each chapter. There's some initial setup required to reset the exercise solutions so they are ready to be solved by you. Use the resetSolutions.sh script to simplify this process. While you're at it, you should also strip out all the anchor comments with the removeAnchors.sh script (these anchors are used for copying code snippets into the book's rendered markdown, and you probably don't need this clutter in your local repo):

cd purescript-book

./scripts/resetSolutions.sh

./scripts/removeAnchors.sh

git add .

git commit --all --message "Exercises ready to be solved"

Now run the tests for this chapter:

cd exercises/chapter2

spago test

You should see the following successful test output:

→ Suite: Euler - Sum of Multiples

✓ Passed: below 10

✓ Passed: below 1000

All 2 tests passed! 🎉

Note that the answer function (found in src/Euler.purs) has been modified to find the multiples of 3 and 5 below any integer. The test suite (found in test/Main.purs) for this answer function is more comprehensive than the test in the earlier getting-started guide. Don't worry about understanding how this test framework code works while reading these early chapters.

The remainder of the book contains lots of exercises. If you write your solutions in the Test.MySolutions module (test/MySolutions.purs), you can check your work against the provided test suite.

Let's work through this next exercise together in test-driven-development style.

## Exercise

1. (Medium) Write a diagonal function to compute the length of the diagonal (or hypotenuse) of a right-angled triangle when given the lengths of the two other sides.

## Solution

We'll start by enabling the tests for this exercise. Move the start of the block-comment down a few lines as shown below. Block comments start with {- and end with -}:

{{#include ../exercises/chapter2/test/Main.purs:diagonalTests}}

{- Move this block comment starting point to enable more tests

If we attempt to run the test now, we'll encounter a compilation error because we have not yet implemented our diagonal function.

$ spago test

Error found:

in module Test.Main

at test/Main.purs:21:27 - 21:35 (line 21, column 27 - line 21, column 35)

Unknown value diagonal

Let's first take a look at what happens with a faulty version of this function. Add the following code to test/MySolutions.purs:

import Data.Number (sqrt)

diagonal w h = sqrt (w \* w + h)

And check our work by running spago test:

→ Suite: diagonal

☠ Failed: 3 4 5 because expected 5.0, got 3.605551275463989

☠ Failed: 5 12 13 because expected 13.0, got 6.082762530298219

2 tests failed:

Uh-oh, that's not quite right. Let's fix this with the correct application of the Pythagorean formula by changing the function to:

{{#include ../exercises/chapter2/test/no-peeking/Solutions.purs:diagonal}}

Trying spago test again now shows all tests are passing:

→ Suite: Euler - Sum of Multiples

✓ Passed: below 10

✓ Passed: below 1000

→ Suite: diagonal

✓ Passed: 3 4 5

✓ Passed: 5 12 13

All 4 tests passed! 🎉

Success! Now you're ready to try these next exercises on your own.

## Exercises

1. (Easy) Write a function circleArea which computes the area of a circle with a given radius. Use the pi constant, which is defined in the Numbers module. Hint: don't forget to import pi by modifying the import Data.Number statement.
2. (Medium) Write a function leftoverCents which takes an Int and returns what's leftover after dividing by 100. Use the rem function. Search [Pursuit](https://pursuit.purescript.org/) for this function to learn about usage and which module to import it from. Note: Your IDE may support auto-importing of this function if you accept the auto-completion suggestion.

## Conclusion

In this chapter, we installed the PureScript compiler and the Spago tool. We also learned how to write solutions to exercises and check these for correctness.

There will be many more exercises in the chapters ahead, and working through those really helps with learning the material. If you're stumped by any of the exercises, please reach out to any of the community resources listed in the [Getting Help](https://book.purescript.org/chapter1.html#getting-help) section of this book, or even file an issue in this [book's repo](https://github.com/purescript-contrib/purescript-book/issues). This reader feedback on which exercises could be made more approachable helps us improve the book.

Once you solve all the exercises in a chapter, you may compare your answers against those in the no-peeking/Solutions.purs. No peeking please without putting in an honest effort to solve these yourself though. And even if you are stuck, try asking a community member for help first, as we would prefer to give you a small hint rather than spoil the exercise. If you found a more elegant solution (that still only requires knowledge of covered content), please send us a PR.

The repo is continuously being revised, so be sure to check for updates before starting each new chapter.

# Functions and Records

## Chapter Goals

This chapter will introduce two building blocks of PureScript programs: functions and records. In addition, we'll see how to structure PureScript programs, and how to use types as an aid to program development.

We will build a simple address book application to manage a list of contacts. This code will introduce some new ideas from the syntax of PureScript.

The front-end of our application will be the interactive mode PSCi, but it would be possible to build on this code to write a front-end in JavaScript. In fact, we will do exactly that in later chapters, adding form validation and save/restore functionality.

## Project Setup

The source code for this chapter is contained in the file src/Data/AddressBook.purs. This file starts with a module declaration and its import list:

{{#include ../exercises/chapter3/src/Data/AddressBook.purs:imports}}

Here, we import several modules:

* The Control.Plus module, which defines the empty value.
* The Data.List module, which is provided by the lists package which can be installed using Spago. It contains a few functions which we will need for working with linked lists.
* The Data.Maybe module, which defines data types and functions for working with optional values.

Notice that the imports for these modules are listed explicitly in parentheses. This is generally a good practice, as it helps to avoid conflicting imports.

Assuming you have cloned the book's source code repository, the project for this chapter can be built using Spago, with the following commands:

$ cd chapter3

$ spago build

## Simple Types

PureScript defines three built-in types which correspond to JavaScript's primitive types: numbers, strings and booleans. These are defined in the Prim module, which is implicitly imported by every module. They are called Number, String, and Boolean, respectively, and you can see them in PSCi by using the :type command to print the types of some simple values:

$ spago repl

> :type 1.0

Number

> :type "test"

String

> :type true

Boolean

PureScript defines some other built-in types: integers, characters, arrays, records, and functions.

Integers are differentiated from floating point values of type Number by the lack of a decimal point:

> :type 1

Int

Character literals are wrapped in single quotes, unlike string literals which use double quotes:

> :type 'a'

Char

Arrays correspond to JavaScript arrays, but unlike in JavaScript, all elements of a PureScript array must have the same type:

> :type [1, 2, 3]

Array Int

> :type [true, false]

Array Boolean

> :type [1, false]

Could not match type Int with type Boolean.

The error in the last example is an error from the type checker, which unsuccessfully attempted to unify (i.e. make equal) the types of the two elements.

Records correspond to JavaScript's objects, and record literals have the same syntax as JavaScript's object literals:

> author = { name: "Phil", interests: ["Functional Programming", "JavaScript"] }

> :type author

{ name :: String

, interests :: Array String

}

This type indicates that the specified object has two fields, a name field which has type String, and an interests field, which has type Array String, i.e. an array of Strings.

Fields of records can be accessed using a dot, followed by the label of the field to access:

> author.name

"Phil"

> author.interests

["Functional Programming","JavaScript"]

PureScript's functions correspond to JavaScript's functions. The PureScript standard libraries provide plenty of examples of functions, and we will see more in this chapter:

> import Prelude

> :type flip

forall a b c. (a -> b -> c) -> b -> a -> c

> :type const

forall a b. a -> b -> a

Functions can be defined at the top-level of a file by specifying arguments before the equals sign:

add :: Int -> Int -> Int

add x y = x + y

Alternatively, functions can be defined inline, by using a backslash character followed by a space-delimited list of argument names. To enter a multi-line declaration in PSCi, we can enter "paste mode" by using the :paste command. In this mode, declarations are terminated using the Control-D key sequence:

> :paste

… add :: Int -> Int -> Int

… add = \x y -> x + y

… ^D

Having defined this function in PSCi, we can apply it to its arguments by separating the two arguments from the function name by whitespace:

> add 10 20

30

## Quantified Types

In the previous section, we saw the types of some functions defined in the Prelude. For example, the flip function had the following type:

> :type flip

forall a b c. (a -> b -> c) -> b -> a -> c

The keyword forall here indicates that flip has a universally quantified type. It means that we can substitute any types for a, b and c, and flip will work with those types.

For example, we might choose the type a to be Int, b to be String and c to be String. In that case we could specialize the type of flip to

(Int -> String -> String) -> String -> Int -> String

We don't have to indicate in code that we want to specialize a quantified type - it happens automatically. For example, we can just use flip as if it had this type already:

> flip (\n s -> show n <> s) "Ten" 10

"10Ten"

While we can choose any types for a, b and c, we have to be consistent. The type of the function we passed to flip had to be consistent with the types of the other arguments. That is why we passed the string "Ten" as the second argument, and the number 10 as the third. It would not work if the arguments were reversed:

> flip (\n s -> show n <> s) 10 "Ten"

Could not match type Int with type String

## Notes On Indentation

PureScript code is indentation-sensitive, just like Haskell, but unlike JavaScript. This means that the whitespace in your code is not meaningless, but rather is used to group regions of code, just like curly braces in C-like languages.

If a declaration spans multiple lines, then any lines except the first must be indented past the indentation level of the first line.

Therefore, the following is valid PureScript code:

add x y z = x +

y + z

But this is not valid code:

add x y z = x +

y + z

In the second case, the PureScript compiler will try to parse two declarations, one for each line.

Generally, any declarations defined in the same block should be indented at the same level. For example, in PSCi, declarations in a let statement must be indented equally. This is valid:

> :paste

… x = 1

… y = 2

… ^D

but this is not:

> :paste

… x = 1

… y = 2

… ^D

Certain PureScript keywords (such as where, of and let) introduce a new block of code, in which declarations must be further-indented:

example x y z = foo + bar

where

foo = x \* y

bar = y \* z

Note how the declarations for foo and bar are indented past the declaration of example.

The only exception to this rule is the where keyword in the initial module declaration at the top of a source file.

## Defining Our Types

A good first step when tackling a new problem in PureScript is to write out type definitions for any values you will be working with. First, let's define a type for records in our address book:

{{#include ../exercises/chapter3/src/Data/AddressBook.purs:Entry}}

This defines a type synonym called Entry - the type Entry is equivalent to the type on the right of the equals symbol: a record type with three fields - firstName, lastName and address. The two name fields will have type String, and the address field will have type Address, defined as follows:

{{#include ../exercises/chapter3/src/Data/AddressBook.purs:Address}}

Note that records can contain other records.

Now let's define a third type synonym, for our address book data structure, which will be represented simply as a linked list of entries:

{{#include ../exercises/chapter3/src/Data/AddressBook.purs:AddressBook}}

Note that List Entry is not the same as Array Entry, which represents an array of entries.

## Type Constructors and Kinds

List is an example of a type constructor. Values do not have the type List directly, but rather List a for some type a. That is, List takes a type argument a and constructs a new type List a.

Note that just like function application, type constructors are applied to other types simply by juxtaposition: the type List Entry is in fact the type constructor List applied to the type Entry - it represents a list of entries.

If we try to incorrectly define a value of type List (by using the type annotation operator ::), we will see a new type of error:

> import Data.List

> Nil :: List

In a type-annotated expression x :: t, the type t must have kind Type

This is a kind error. Just like values are distinguished by their types, types are distinguished by their kinds, and just like ill-typed values result in type errors, ill-kinded types result in kind errors.

There is a special kind called Type which represents the kind of all types which have values, like Number and String.

There are also kinds for type constructors. For example, the kind Type -> Type represents a function from types to types, just like List. So the error here occurred because values are expected to have types with kind Type, but List has kind Type -> Type.

To find out the kind of a type, use the :kind command in PSCi. For example:

> :kind Number

Type

> import Data.List

> :kind List

Type -> Type

> :kind List String

Type

PureScript's kind system supports other interesting kinds, which we will see later in the book.

## Displaying Address Book Entries

Let's write our first function, which will render an address book entry as a string. We start by giving the function a type. This is optional, but good practice, since it acts as a form of documentation. In fact, the PureScript compiler will give a warning if a top-level declaration does not contain a type annotation. A type declaration separates the name of a function from its type with the :: symbol:

{{#include ../exercises/chapter3/src/Data/AddressBook.purs:showEntry\_signature}}

This type signature says that showEntry is a function, which takes an Entry as an argument and returns a String. Here is the code for showEntry:

{{#include ../exercises/chapter3/src/Data/AddressBook.purs:showEntry\_implementation}}

This function concatenates the three fields of the Entry record into a single string, using the showAddress function to turn the record inside the address field into a String. showAddress is defined similarly:

{{#include ../exercises/chapter3/src/Data/AddressBook.purs:showAddress}}

A function definition begins with the name of the function, followed by a list of argument names. The result of the function is specified after the equals sign. Fields are accessed with a dot, followed by the field name. In PureScript, string concatenation uses the diamond operator (<>), instead of the plus operator like in JavaScript.

## Test Early, Test Often

The PSCi interactive mode allows for rapid prototyping with immediate feedback, so let's use it to verify that our first few functions behave as expected.

First, build the code you've written:

$ spago build

Next, load PSCi, and use the import command to import your new module:

$ spago repl

> import Data.AddressBook

We can create an entry by using a record literal, which looks just like an anonymous object in JavaScript.

> address = { street: "123 Fake St.", city: "Faketown", state: "CA" }

Now, try applying our function to the example:

> showAddress address

"123 Fake St., Faketown, CA"

Let's also test showEntry by creating an address book entry record containing our example address:

> entry = { firstName: "John", lastName: "Smith", address: address }

> showEntry entry

"Smith, John: 123 Fake St., Faketown, CA"

## Creating Address Books

Now let's write some utility functions for working with address books. We will need a value which represents an empty address book: an empty list.

{{#include ../exercises/chapter3/src/Data/AddressBook.purs:emptyBook}}

We will also need a function for inserting a value into an existing address book. We will call this function insertEntry. Start by giving its type:

{{#include ../exercises/chapter3/src/Data/AddressBook.purs:insertEntry\_signature}}

This type signature says that insertEntry takes an Entry as its first argument, and an AddressBook as a second argument, and returns a new AddressBook.

We don't modify the existing AddressBook directly. Instead, we return a new AddressBook which contains the same data. As such, AddressBook is an example of an immutable data structure. This is an important idea in PureScript - mutation is a side-effect of code, and inhibits our ability to reason effectively about its behavior, so we prefer pure functions and immutable data where possible.

To implement insertEntry, we can use the Cons function from Data.List. To see its type, open PSCi and use the :type command:

$ spago repl

> import Data.List

> :type Cons

forall a. a -> List a -> List a

This type signature says that Cons takes a value of some type a, and a list of elements of type a, and returns a new list with entries of the same type. Let's specialize this with a as our Entry type:

Entry -> List Entry -> List Entry

But List Entry is the same as AddressBook, so this is equivalent to

Entry -> AddressBook -> AddressBook

In our case, we already have the appropriate inputs: an Entry, and an AddressBook, so can apply Cons and get a new AddressBook, which is exactly what we wanted!

Here is our implementation of insertEntry:

insertEntry entry book = Cons entry book

This brings the two arguments entry and book into scope, on the left hand side of the equals symbol, and then applies the Cons function to create the result.

## Curried Functions

Functions in PureScript take exactly one argument. While it looks like the insertEntry function takes two arguments, it is in fact an example of a curried function.

The -> operator in the type of insertEntry associates to the right, which means that the compiler parses the type as

Entry -> (AddressBook -> AddressBook)

That is, insertEntry is a function which returns a function! It takes a single argument, an Entry, and returns a new function, which in turn takes a single AddressBook argument and returns a new AddressBook.

This means that we can partially apply insertEntry by specifying only its first argument, for example. In PSCi, we can see the result type:

> :type insertEntry entry

AddressBook -> AddressBook

As expected, the return type was a function. We can apply the resulting function to a second argument:

> :type (insertEntry entry) emptyBook

AddressBook

Note though that the parentheses here are unnecessary - the following is equivalent:

> :type insertEntry entry emptyBook

AddressBook

This is because function application associates to the left, and this explains why we can just specify function arguments one after the other, separated by whitespace.

The -> operator in function types is a type constructor for functions. It takes two type arguments, the function's argument type and the return type. The left and right operands respectively.

Note that in the rest of the book, I will talk about things like "functions of two arguments". However, it is to be understood that this means a curried function, taking a first argument and returning a function that takes the second.

Now consider the definition of insertEntry:

insertEntry :: Entry -> AddressBook -> AddressBook

insertEntry entry book = Cons entry book

If we explicitly parenthesize the right-hand side, we get (Cons entry) book. That is, insertEntry entry is a function whose argument is just passed along to the (Cons entry) function. But if two functions have the same result for every input, then they are the same function! So we can remove the argument book from both sides:

insertEntry :: Entry -> AddressBook -> AddressBook

insertEntry entry = Cons entry

But now, by the same argument, we can remove entry from both sides:

{{#include ../exercises/chapter3/src/Data/AddressBook.purs:insertEntry}}

This process is called eta conversion, and can be used (along with some other techniques) to rewrite functions in point-free form, which means functions defined without reference to their arguments.

In the case of insertEntry, eta conversion has resulted in a very clear definition of our function - "insertEntry is just cons on lists". However, it is arguable whether point-free form is better in general.

## Property Accessors

One common pattern is to use a function to access individual fields (or "properties") of a record. An inline function to extract an Address from an Entry could be written as:

\entry -> entry.address

PureScript also allows [property accessor](https://github.com/purescript/documentation/blob/master/language/Syntax.md#property-accessors) shorthand, where an underscore acts as the anonymous function argument, so the inline function above is equivalent to:

\_.address

This works with any number of levels or properties, so a function to extract the city associated with an Entry could be written as:

\_.address.city

For example:

> address = { street: "123 Fake St.", city: "Faketown", state: "CA" }

> entry = { firstName: "John", lastName: "Smith", address: address }

> \_.lastName entry

"Smith"

> \_.address.city entry

"Faketown"

## Querying the Address Book

The last function we need to implement for our minimal address book application will look up a person by name and return the correct Entry. This will be a nice application of building programs by composing small functions - a key idea from functional programming.

We can first filter the address book, keeping only those entries with the correct first and last names. Then we can simply return the head (i.e. first) element of the resulting list.

With this high-level specification of our approach, we can calculate the type of our function. First open PSCi, and find the types of the filter and head functions:

$ spago repl

> import Data.List

> :type filter

forall a. (a -> Boolean) -> List a -> List a

> :type head

forall a. List a -> Maybe a

Let's pick apart these two types to understand their meaning.

filter is a curried function of two arguments. Its first argument is a function, which takes an element of the list and returns a Boolean value as a result. Its second argument is a list of elements, and the return value is another list.

head takes a list as its argument, and returns a type we haven't seen before: Maybe a. Maybe a represents an optional value of type a, and provides a type-safe alternative to using null to indicate a missing value in languages like JavaScript. We will see it again in more detail in later chapters.

The universally quantified types of filter and head can be specialized by the PureScript compiler, to the following types:

filter :: (Entry -> Boolean) -> AddressBook -> AddressBook

head :: AddressBook -> Maybe Entry

We know that we will need to pass the first and last names that we want to search for, as arguments to our function.

We also know that we will need a function to pass to filter. Let's call this function filterEntry. filterEntry will have type Entry -> Boolean. The application filter filterEntry will then have type AddressBook -> AddressBook. If we pass the result of this function to the head function, we get our result of type Maybe Entry.

Putting these facts together, a reasonable type signature for our function, which we will call findEntry, is:

{{#include ../exercises/chapter3/src/Data/AddressBook.purs:findEntry\_signature}}

This type signature says that findEntry takes two strings, the first and last names, and a AddressBook, and returns an optional Entry. The optional result will contain a value only if the name is found in the address book.

And here is the definition of findEntry:

findEntry firstName lastName book = head (filter filterEntry book)

where

filterEntry :: Entry -> Boolean

filterEntry entry = entry.firstName == firstName && entry.lastName == lastName

Let's go over this code step by step.

findEntry brings three names into scope: firstName and lastName, both representing strings, and book, an AddressBook.

The right hand side of the definition combines the filter and head functions: first, the list of entries is filtered, and the head function is applied to the result.

The predicate function filterEntry is defined as an auxiliary declaration inside a where clause. This way, the filterEntry function is available inside the definition of our function, but not outside it. Also, it can depend on the arguments to the enclosing function, which is essential here because filterEntry uses the firstName and lastName arguments to filter the specified Entry.

Note that, just like for top-level declarations, it was not necessary to specify a type signature for filterEntry. However, doing so is recommended as a form of documentation.

## Infix Function Application

Most of the functions discussed so far used prefix function application, where the function name was put before the arguments. For example, when using the insertEntry function to add an Entry (john) to an empty AddressBook, we might write:

> book1 = insertEntry john emptyBook

However, this chapter has also included examples of infix [binary operators](https://github.com/purescript/documentation/blob/master/language/Syntax.md#binary-operators), such as the == operator in the definition of filterEntry, where the operator is put between the two arguments. These infix operators are actually defined in the PureScript source as infix aliases for their underlying prefix implementations. For example, == is defined as an infix alias for the prefix eq function with the line:

infix 4 eq as ==

and therefore entry.firstName == firstName in filterEntry could be replaced with the eq entry.firstName firstName. We'll cover a few more examples of defining infix operators later in this section.

There are situations where putting a prefix function in an infix position as an operator leads to more readable code. One example is the mod function:

> mod 8 3

2

The above usage works fine, but is awkward to read. A more familiar phrasing is "eight mod three", which you can achieve by wrapping a prefix function in backticks (`):

> 8 `mod` 3

2

In the same way, wrapping insertEntry in backticks turns it into an infix operator, such that book1 and book2 below are equivalent:

book1 = insertEntry john emptyBook

book2 = john `insertEntry` emptyBook

We can make an AddressBook with multiple entries by using multiple applications of insertEntry as a prefix function (book3) or as an infix operator (book4) as shown below:

book3 = insertEntry john (insertEntry peggy (insertEntry ned emptyBook))

book4 = john `insertEntry` (peggy `insertEntry` (ned `insertEntry` emptyBook))

We can also define an infix operator alias (or synonym) for insertEntry. We'll arbitrarily choose ++ for this operator, give it a [precedence](https://github.com/purescript/documentation/blob/master/language/Syntax.md#precedence) of 5, and make it right [associative](https://github.com/purescript/documentation/blob/master/language/Syntax.md#associativity) using infixr:

infixr 5 insertEntry as ++

This new operator lets us rewrite the above book4 example as:

book5 = john ++ (peggy ++ (ned ++ emptyBook))

and the right associativity of our new ++ operator lets us get rid of the parentheses without changing the meaning:

book6 = john ++ peggy ++ ned ++ emptyBook

Another common technique for eliminating parens is to use apply's infix operator $, along with your standard prefix functions.

For example, the earlier book3 example could be rewritten as:

book7 = insertEntry john $ insertEntry peggy $ insertEntry ned emptyBook

Substituting $ for parens is usually easier to type and (arguably) easier to read. A mnemonic to remember the meaning of this symbol is to think of the dollar sign as being drawn from two parens that are also being crossed-out, suggesting the parens are now unnecessary.

Note that $ isn't special syntax that's hardcoded into the language. It's simply the infix operator for a regular function called apply, which is defined in Data.Function as follows:

apply :: forall a b. (a -> b) -> a -> b

apply f x = f x

infixr 0 apply as $

The apply function takes another function (of type (a -> b)) as its first argument and a value (of type a) as its second argument, then calls that function with that value. If it seems like this function doesn't contribute anything meaningful, you are absolutely correct! Your program is logically identical without it (see [referential transparency](https://en.wikipedia.org/wiki/Referential_transparency)). The syntactic utility of this function comes from the special properties assigned to its infix operator. $ is a right-associative (infixr), low precedence (0) operator, which lets us remove sets of parentheses for deeply-nested applications.

Another parens-busting opportunity for the $ operator is in our earlier findEntry function:

findEntry firstName lastName book = head $ filter filterEntry book

We'll see an even more elegant way to rewrite this line with "function composition" in the next section.

If you'd like to use a concise infix operator alias as a prefix function, you can surround it in parentheses:

> 8 + 3

11

> (+) 8 3

11

Alternatively, operators can be partially applied by surrounding the expression with parentheses and using \_ as an operand in an [operator section](https://github.com/purescript/documentation/blob/master/language/Syntax.md#operator-sections). You can think of this as a more convenient way to create simple anonymous functions (although in the below example, we're then binding that anonymous function to a name, so it's not so anonymous anymore):

> add3 = (3 + \_)

> add3 2

5

To summarize, the following are equivalent definitions of a function that adds 5 to its argument:

add5 x = 5 + x

add5 x = add 5 x

add5 x = (+) 5 x

add5 x = 5 `add` x

add5 = add 5

add5 = \x -> 5 + x

add5 = (5 + \_)

add5 x = 5 `(+)` x -- Yo Dawg, I herd you like infix, so we put infix in your infix!

## Function Composition

Just like we were able to simplify the insertEntry function by using eta conversion, we can simplify the definition of findEntry by reasoning about its arguments.

Note that the book argument is passed to the filter filterEntry function, and the result of this application is passed to head. In other words, book is passed to the composition of the functions filter filterEntry and head.

In PureScript, the function composition operators are <<< and >>>. The first is "backwards composition", and the second is "forwards composition".

We can rewrite the right-hand side of findEntry using either operator. Using backwards-composition, the right-hand side would be

(head <<< filter filterEntry) book

In this form, we can apply the eta conversion trick from earlier, to arrive at the final form of findEntry:

{{#include ../exercises/chapter3/src/Data/AddressBook.purs:findEntry\_implementation}}

...

An equally valid right-hand side would be:

filter filterEntry >>> head

Either way, this gives a clear definition of the findEntry function: "findEntry is the composition of a filtering function and the head function".

I will let you make your own decision which definition is easier to understand, but it is often useful to think of functions as building blocks in this way - each function executing a single task, and solutions assembled using function composition.

## Exercises

1. (Easy) Test your understanding of the findEntry function by writing down the types of each of its major subexpressions. For example, the type of the head function as used is specialized to AddressBook -> Maybe Entry. Note: There is no test for this exercise.
2. (Medium) Write a function findEntryByStreet :: String -> AddressBook -> Maybe Entry which looks up an Entry given a street address. Hint reusing the existing code in findEntry. Test your function in PSCi and by running spago test.
3. (Medium) Rewrite findEntryByStreet to replace filterEntry with the composition (using <<< or >>>) of: a property accessor (using the \_. notation); and a function that tests whether its given string argument is equal to the given street address.
4. (Medium) Write a function isInBook which tests whether a name appears in a AddressBook, returning a Boolean value. Hint: Use PSCi to find the type of the Data.List.null function, which tests whether a list is empty or not.
5. (Difficult) Write a function removeDuplicates which removes "duplicate" address book entries. We'll consider entries duplicated if they share the same first and last names, while ignoring address fields. Hint: Use PSCi to find the type of the Data.List.nubByEq function, which removes duplicate elements from a list based on an equality predicate. Note that the first element in each set of duplicates (closest to list head) is the one that is kept.

## Conclusion

In this chapter, we covered several new functional programming concepts:

* How to use the interactive mode PSCi to experiment with functions and test ideas.
* The role of types as both a correctness tool, and an implementation tool.
* The use of curried functions to represent functions of multiple arguments.
* Creating programs from smaller components by composition.
* Structuring code neatly using where expressions.
* How to avoid null values by using the Maybe type.
* Using techniques like eta conversion and function composition to refactor code into a clear specification.

In the following chapters, we'll build on these ideas.

# Recursion, Maps And Folds

## Chapter Goals

In this chapter, we will look at how recursive functions can be used to structure algorithms. Recursion is a basic technique used in functional programming, which we will use throughout this book.

We will also cover some standard functions from PureScript's standard libraries. We will see the map and fold functions, as well as some useful special cases, like filter and concatMap.

The motivating example for this chapter is a library of functions for working with a virtual filesystem. We will apply the techniques learned in this chapter to write functions which compute properties of the files represented by a model of a filesystem.

## Project Setup

The source code for this chapter is contained in src/Data/Path.purs and test/Examples.purs. The Data.Path module contains a model of a virtual filesystem. You do not need to modify the contents of this module. Implement your solutions to the exercises in the Test.MySolutions module. Enable accompanying tests in the Test.Main module as you complete each exercise and check your work by running spago test.

The project has the following dependencies:

* maybe, which defines the Maybe type constructor
* arrays, which defines functions for working with arrays
* strings, which defines functions for working with JavaScript strings
* foldable-traversable, which defines functions for folding arrays and other data structures
* console, which defines functions for printing to the console

## Introduction

Recursion is an important technique in programming in general, but particularly common in pure functional programming, because, as we will see in this chapter, recursion helps to reduce the mutable state in our programs.

Recursion is closely linked to the divide and conquer strategy: to solve a problem on certain inputs, we can break down the inputs into smaller parts, solve the problem on those parts, and then assemble a solution from the partial solutions.

Let's see some simple examples of recursion in PureScript.

Here is the usual factorial function example:

{{#include ../exercises/chapter4/test/Examples.purs:factorial}}

Here, we can see how the factorial function is computed by reducing the problem to a subproblem - that of computing the factorial of a smaller integer. When we reach zero, the answer is immediate.

Here is another common example, which computes the Fibonacci function:

{{#include ../exercises/chapter4/test/Examples.purs:fib}}

Again, this problem is solved by considering the solutions to subproblems. In this case, there are two subproblems, corresponding to the expressions fib (n - 1) and fib (n - 2). When these two subproblems are solved, we assemble the result by adding the partial results.

Note that, while the above examples of factorial and fib work as intended, a more idiomatic implementation would use pattern matching instead of if/then/else. Pattern matching techniques are discussed in a later chapter.

## Recursion on Arrays

We are not limited to defining recursive functions over the Int type! We will see recursive functions defined over a wide array of data types when we cover pattern matching later in the book, but for now, we will restrict ourselves to numbers and arrays.

Just as we branch based on whether the input is non-zero, in the array case, we will branch based on whether the input is non-empty. Consider this function, which computes the length of an array using recursion:

import Prelude

import Data.Array (null, tail)

import Data.Maybe (fromMaybe)

{{#include ../exercises/chapter4/test/Examples.purs:length}}

In this function, we use an if .. then .. else expression to branch based on the emptiness of the array. The null function returns true on an empty array. Empty arrays have length zero, and a non-empty array has a length that is one more than the length of its tail.

The tail function returns a Maybe wrapping the given array without its first element. If the array is empty (i.e. it doesn't have a tail) Nothing is returned. The fromMaybe function takes a default value and a Maybe value. If the latter is Nothing it returns the default, in the other case it returns the value wrapped by Just.

This example is obviously a very impractical way to find the length of an array in JavaScript, but should provide enough help to allow you to complete the following exercises:

## Exercises

1. (Easy) Write a recursive function isEven which returns true if and only if its input is an even integer.
2. (Medium) Write a recursive function countEven which counts the number of even integers in an array. Hint: the function head (also available in Data.Array) can be used to find the first element in a non-empty array.

## Maps

The map function is an example of a recursive function on arrays. It is used to transform the elements of an array by applying a function to each element in turn. Therefore, it changes the contents of the array, but preserves its shape (i.e. its length).

When we cover type classes later in the book we will see that the map function is an example of a more general pattern of shape-preserving functions which transform a class of type constructors called functors.

Let's try out the map function in PSCi:

$ spago repl

> import Prelude

> map (\n -> n + 1) [1, 2, 3, 4, 5]

[2, 3, 4, 5, 6]

Notice how map is used - we provide a function which should be "mapped over" the array in the first argument, and the array itself in its second.

## Infix Operators

The map function can also be written between the mapping function and the array, by wrapping the function name in backticks:

> (\n -> n + 1) `map` [1, 2, 3, 4, 5]

[2, 3, 4, 5, 6]

This syntax is called infix function application, and any function can be made infix in this way. It is usually most appropriate for functions with two arguments.

There is an operator which is equivalent to the map function when used with arrays, called <$>. This operator can be used infix like any other binary operator:

> (\n -> n + 1) <$> [1, 2, 3, 4, 5]

[2, 3, 4, 5, 6]

Let's look at the type of map:

> :type map

forall a b f. Functor f => (a -> b) -> f a -> f b

The type of map is actually more general than we need in this chapter. For our purposes, we can treat map as if it had the following less general type:

forall a b. (a -> b) -> Array a -> Array b

This type says that we can choose any two types, a and b, with which to apply the map function. a is the type of elements in the source array, and b is the type of elements in the target array. In particular, there is no reason why map has to preserve the type of the array elements. We can use map or <$> to transform integers to strings, for example:

> show <$> [1, 2, 3, 4, 5]

["1","2","3","4","5"]

Even though the infix operator <$> looks like special syntax, it is in fact just an alias for a regular PureScript function. The function is simply applied using infix syntax. In fact, the function can be used like a regular function by enclosing its name in parentheses. This means that we can used the parenthesized name (<$>) in place of map on arrays:

> (<$>) show [1, 2, 3, 4, 5]

["1","2","3","4","5"]

Infix function names are defined as aliases for existing function names. For example, the Data.Array module defines an infix operator (..) as a synonym for the range function, as follows:

infix 8 range as ..

We can use this operator as follows:

> import Data.Array

> 1 .. 5

[1, 2, 3, 4, 5]

> show <$> (1 .. 5)

["1","2","3","4","5"]

Note: Infix operators can be a great tool for defining domain-specific languages with a natural syntax. However, used excessively, they can render code unreadable to beginners, so it is wise to exercise caution when defining any new operators.

In the example above, we parenthesized the expression 1 .. 5, but this was actually not necessary, because the Data.Array module assigns a higher precedence level to the .. operator than that assigned to the <$> operator. In the example above, the precedence of the .. operator was defined as 8, the number after the infix keyword. This is higher than the precedence level of <$>, meaning that we do not need to add parentheses:

> show <$> 1 .. 5

["1","2","3","4","5"]

If we wanted to assign an associativity (left or right) to an infix operator, we could do so with the infixl and infixr keywords instead. Using infix assigns no associativity, meaning that you must parenthesize any expression using the same operator multiple times or using multiple operators of the same precedence.

## Filtering Arrays

The Data.Array module provides another function filter, which is commonly used together with map. It provides the ability to create a new array from an existing array, keeping only those elements which match a predicate function.

For example, suppose we wanted to compute an array of all numbers between 1 and 10 which were even. We could do so as follows:

> import Data.Array

> filter (\n -> n `mod` 2 == 0) (1 .. 10)

[2,4,6,8,10]

## Exercises

1. (Easy) Write a function squared which calculates the squares of an array of numbers. Hint: Use the map or <$> function.
2. (Easy) Write a function keepNonNegative which removes the negative numbers from an array of numbers. Hint: Use the filter function.
3. (Medium)
   * Define an infix synonym <$?> for filter. Note: Infix synonyms may not be defined in the REPL, but you can define it in a file.
   * Write a keepNonNegativeRewrite function, which is the same as keepNonNegative, but replaces filter with your new infix operator <$?>.
   * Experiment with the precedence level and associativity of your operator in PSCi. Note: There are no unit tests for this step.

## Flattening Arrays

Another standard function on arrays is the concat function, defined in Data.Array. concat flattens an array of arrays into a single array:

> import Data.Array

> :type concat

forall a. Array (Array a) -> Array a

> concat [[1, 2, 3], [4, 5], [6]]

[1, 2, 3, 4, 5, 6]

There is a related function called concatMap which is like a combination of the concat and map functions. Where map takes a function from values to values (possibly of a different type), concatMap takes a function from values to arrays of values.

Let's see it in action:

> import Data.Array

> :type concatMap

forall a b. (a -> Array b) -> Array a -> Array b

> concatMap (\n -> [n, n \* n]) (1 .. 5)

[1,1,2,4,3,9,4,16,5,25]

Here, we call concatMap with the function \n -> [n, n \* n] which sends an integer to the array of two elements consisting of that integer and its square. The result is an array of ten integers: the integers from 1 to 5 along with their squares.

Note how concatMap concatenates its results. It calls the provided function once for each element of the original array, generating an array for each. Finally, it collapses all of those arrays into a single array, which is its result.

map, filter and concatMap form the basis for a whole range of functions over arrays called "array comprehensions".

## Array Comprehensions

Suppose we wanted to find the factors of a number n. One simple way to do this would be by brute force: we could generate all pairs of numbers between 1 and n, and try multiplying them together. If the product was n, we would have found a pair of factors of n.

We can perform this computation using an array comprehension. We will do so in steps, using PSCi as our interactive development environment.

The first step is to generate an array of pairs of numbers below n, which we can do using concatMap.

Let's start by mapping each number to the array 1 .. n:

> pairs n = concatMap (\i -> 1 .. n) (1 .. n)

We can test our function

> pairs 3

[1,2,3,1,2,3,1,2,3]

This is not quite what we want. Instead of just returning the second element of each pair, we need to map a function over the inner copy of 1 .. n which will allow us to keep the entire pair:

> :paste

… pairs' n =

… concatMap (\i ->

… map (\j -> [i, j]) (1 .. n)

… ) (1 .. n)

… ^D

> pairs' 3

[[1,1],[1,2],[1,3],[2,1],[2,2],[2,3],[3,1],[3,2],[3,3]]

This is looking better. However, we are generating too many pairs: we keep both [1, 2] and [2, 1] for example. We can exclude the second case by making sure that j only ranges from i to n:

> :paste

… pairs'' n =

… concatMap (\i ->

… map (\j -> [i, j]) (i .. n)

… ) (1 .. n)

… ^D

> pairs'' 3

[[1,1],[1,2],[1,3],[2,2],[2,3],[3,3]]

Great! Now that we have all of the pairs of potential factors, we can use filter to choose the pairs which multiply to give n:

> import Data.Foldable

> factors n = filter (\pair -> product pair == n) (pairs'' n)

> factors 10

[[1,10],[2,5]]

This code uses the product function from the Data.Foldable module in the foldable-traversable library.

Excellent! We've managed to find the correct set of factor pairs without duplicates.

## Do Notation

However, we can improve the readability of our code considerably. map and concatMap are so fundamental, that they (or rather, their generalizations map and bind) form the basis of a special syntax called do notation.

Note: Just like map and concatMap allowed us to write array comprehensions, the more general operators map and bind allow us to write so-called monad comprehensions. We'll see plenty more examples of monads later in the book, but in this chapter, we will only consider arrays.

We can rewrite our factors function using do notation as follows:

{{#include ../exercises/chapter4/test/Examples.purs:factors}}

The keyword do introduces a block of code which uses do notation. The block consists of expressions of a few types:

* Expressions which bind elements of an array to a name. These are indicated with the backwards-facing arrow <-, with a name on the left, and an expression on the right whose type is an array.
* Expressions which do not bind elements of the array to names. The do result is an example of this kind of expression and is illustrated in the last line, pure [i, j].
* Expressions which give names to expressions, using the let keyword.

This new notation hopefully makes the structure of the algorithm clearer. If you mentally replace the arrow <- with the word "choose", you might read it as follows: "choose an element i between 1 and n, then choose an element j between i and n, and return [i, j]".

In the last line, we use the pure function. This function can be evaluated in PSCi, but we have to provide a type:

> pure [1, 2] :: Array (Array Int)

[[1, 2]]

In the case of arrays, pure simply constructs a singleton array. In fact, we could modify our factors function to use this form, instead of using pure:

{{#include ../exercises/chapter4/test/Examples.purs:factorsV2}}

and the result would be the same.

## Guards

One further change we can make to the factors function is to move the filter inside the array comprehension. This is possible using the guard function from the Control.Alternative module (from the control package):

import Control.Alternative (guard)

{{#include ../exercises/chapter4/test/Examples.purs:factorsV3}}

Just like pure, we can apply the guard function in PSCi to understand how it works. The type of the guard function is more general than we need here:

> import Control.Alternative

> :type guard

forall m. Alternative m => Boolean -> m Unit

In our case, we can assume that PSCi reported the following type:

Boolean -> Array Unit

For our purposes, the following calculations tell us everything we need to know about the guard function on arrays:

> import Data.Array

> length $ guard true

1

> length $ guard false

0

That is, if guard is passed an expression which evaluates to true, then it returns an array with a single element. If the expression evaluates to false, then its result is empty.

This means that if the guard fails, then the current branch of the array comprehension will terminate early with no results. This means that a call to guard is equivalent to using filter on the intermediate array. Depending on the application, you might prefer to use guard instead of a filter. Try the two definitions of factors to verify that they give the same results.

## Exercises

1. (Easy) Write a function isPrime which tests if its integer argument is prime or not. Hint: Use the factors function.
2. (Medium) Write a function cartesianProduct which uses do notation to find the cartesian product of two arrays, i.e. the set of all pairs of elements a, b, where a is an element of the first array, and b is an element of the second.
3. (Medium) Write a function triples :: Int -> Array (Array Int) which takes a number n and returns all Pythagorean triples whose components (the a, b and c values) are each less than or equal to n. A Pythagorean triple is an array of numbers [a, b, c] such that a² + b² = c². Hint: Use the guard function in an array comprehension.
4. (Difficult) Write a function primeFactors which produces the [prime factorization](https://www.mathsisfun.com/prime-factorization.html) of n, i.e. the array of prime integers whose product is n. Hint: for an integer greater than 1, break the problem down into two subproblems: finding the first factor, and finding the remaining factors.

## Folds

Left and right folds over arrays provide another class of interesting functions which can be implemented using recursion.

Start by importing the Data.Foldable module, and inspecting the types of the foldl and foldr functions using PSCi:

> import Data.Foldable

> :type foldl

forall a b f. Foldable f => (b -> a -> b) -> b -> f a -> b

> :type foldr

forall a b f. Foldable f => (a -> b -> b) -> b -> f a -> b

These types are actually more general than we are interested in right now. For the purposes of this chapter, we can assume that PSCi had given the following (more specific) answer:

> :type foldl

forall a b. (b -> a -> b) -> b -> Array a -> b

> :type foldr

forall a b. (a -> b -> b) -> b -> Array a -> b

In both of these cases, the type a corresponds to the type of elements of our array. The type b can be thought of as the type of an "accumulator", which will accumulate a result as we traverse the array.

The difference between the foldl and foldr functions is the direction of the traversal. foldl folds the array "from the left", whereas foldr folds the array "from the right".

Let's see these functions in action. Let's use foldl to sum an array of integers. The type a will be Int, and we can also choose the result type b to be Int. We need to provide three arguments: a function Int -> Int -> Int, which will add the next element to the accumulator, an initial value for the accumulator of type Int, and an array of Ints to add. For the first argument, we can just use the addition operator, and the initial value of the accumulator will be zero:

> foldl (+) 0 (1 .. 5)

15

In this case, it didn't matter whether we used foldl or foldr, because the result is the same, no matter what order the additions happen in:

> foldr (+) 0 (1 .. 5)

15

Let's write an example where the choice of folding function does matter, in order to illustrate the difference. Instead of the addition function, let's use string concatenation to build a string:

> foldl (\acc n -> acc <> show n) "" [1,2,3,4,5]

"12345"

> foldr (\n acc -> acc <> show n) "" [1,2,3,4,5]

"54321"

This illustrates the difference between the two functions. The left fold expression is equivalent to the following application:

((((("" <> show 1) <> show 2) <> show 3) <> show 4) <> show 5)

whereas the right fold is equivalent to this:

((((("" <> show 5) <> show 4) <> show 3) <> show 2) <> show 1)

## Tail Recursion

Recursion is a powerful technique for specifying algorithms, but comes with a problem: evaluating recursive functions in JavaScript can lead to stack overflow errors if our inputs are too large.

It is easy to verify this problem, with the following code in PSCi:

> :paste

… f n =

… if n == 0

… then 0

… else 1 + f (n - 1)

… ^D

> f 10

10

> f 100000

RangeError: Maximum call stack size exceeded

This is a problem. If we are going to adopt recursion as a standard technique from functional programming, then we need a way to deal with possibly unbounded recursion.

PureScript provides a partial solution to this problem in the form of tail recursion optimization.

Note: more complete solutions to the problem can be implemented in libraries using so-called trampolining, but that is beyond the scope of this chapter. The interested reader can consult the documentation for the [free](https://pursuit.purescript.org/packages/purescript-free) and [tailrec](https://pursuit.purescript.org/packages/purescript-tailrec) packages.

The key observation which enables tail recursion optimization is the following: a recursive call in tail position to a function can be replaced with a jump, which does not allocate a stack frame. A call is in tail position when it is the last call made before a function returns. This is the reason why we observed a stack overflow in the example - the recursive call to f was not in tail position.

In practice, the PureScript compiler does not replace the recursive call with a jump, but rather replaces the entire recursive function with a while loop.

Here is an example of a recursive function with all recursive calls in tail position:

{{#include ../exercises/chapter4/test/Examples.purs:factorialTailRec}}

Notice that the recursive call to factorialTailRec is the last thing that happens in this function - it is in tail position.

## Accumulators

One common way to turn a function which is not tail recursive into a tail recursive function is to use an accumulator parameter. An accumulator parameter is an additional parameter which is added to a function which accumulates a return value, as opposed to using the return value to accumulate the result.

For example, consider again the length function presented in the beginning of the chapter:

length :: forall a. Array a -> Int

length arr =

if null arr

then 0

else 1 + (length $ fromMaybe [] $ tail arr)

This implementation is not tail recursive, so the generated JavaScript will cause a stack overflow when executed on a large input array. However, we can make it tail recursive, by introducing a second function argument to accumulate the result instead:

{{#include ../exercises/chapter4/test/Examples.purs:lengthTailRec}}

In this case, we delegate to the helper function length', which is tail recursive - its only recursive call is in the last case, and is in tail position. This means that the generated code will be a while loop, and will not blow the stack for large inputs.

To understand the implementation of lengthTailRec, note that the helper function length' essentially uses the accumulator parameter to maintain an additional piece of state - the partial result. It starts out at 0, and grows by adding 1 for every element in the input array.

Note also that while we might think of the accumulator as "state", there is no direct mutation going on.

## Prefer Folds to Explicit Recursion

If we can write our recursive functions using tail recursion, then we can benefit from tail recursion optimization, so it becomes tempting to try to write all of our functions in this form. However, it is often easy to forget that many functions can be written directly as a fold over an array or similar data structure. Writing algorithms directly in terms of combinators such as map and fold has the added advantage of code simplicity - these combinators are well-understood, and as such, communicate the intent of the algorithm much better than explicit recursion.

For example, we can reverse an array using foldr:

> import Data.Foldable

> :paste

… reverse :: forall a. Array a -> Array a

… reverse = foldr (\x xs -> xs <> [x]) []

… ^D

> reverse [1, 2, 3]

[3,2,1]

Writing reverse in terms of foldl will be left as an exercise for the reader.

## Exercises

1. (Easy) Write a function allTrue which uses foldl to test whether an array of boolean values are all true.
2. (Medium - No Test) Characterize those arrays xs for which the function foldl (==) false xs returns true. In other words, complete the sentence: "The function returns true when xs contains ..."
3. (Medium) Write a function fibTailRec which is the same as fib but in tail recursive form. Hint: Use an accumulator parameter.
4. (Medium) Write reverse in terms of foldl.

## A Virtual Filesystem

In this section, we're going to apply what we've learned, writing functions which will work with a model of a filesystem. We will use maps, folds and filters to work with a predefined API.

The Data.Path module defines an API for a virtual filesystem, as follows:

* There is a type Path which represents a path in the filesystem.
* There is a path root which represents the root directory.
* The ls function enumerates the files in a directory.
* The filename function returns the file name for a Path.
* The size function returns the file size for a Path which represents a file.
* The isDirectory function tests whether a Path is a file or a directory.

In terms of types, we have the following type definitions:

root :: Path

ls :: Path -> Array Path

filename :: Path -> String

size :: Path -> Maybe Int

isDirectory :: Path -> Boolean

We can try out the API in PSCi:

$ spago repl

> import Data.Path

> root

/

> isDirectory root

true

> ls root

[/bin/,/etc/,/home/]

The Test.Examples module defines functions which use the Data.Path API. You do not need to modify the Data.Path module, or understand its implementation. We will work entirely in the Test.Examples module.

## Listing All Files

Let's write a function which performs a deep enumeration of all files inside a directory. This function will have the following type:

{{#include ../exercises/chapter4/test/Examples.purs:allFiles\_signature}}

We can define this function by recursion. First, we can use ls to enumerate the immediate children of the directory. For each child, we can recursively apply allFiles, which will return an array of paths. concatMap will allow us to apply allFiles and flatten the results at the same time.

Finally, we use the cons operator : to include the current file:

{{#include ../exercises/chapter4/test/Examples.purs:allFiles\_implementation}}

Note: the cons operator : actually has poor performance on immutable arrays, so it is not recommended in general. Performance can be improved by using other data structures, such as linked lists and sequences.

Let's try this function in PSCi:

> import Test.Examples

> import Data.Path

> allFiles root

[/,/bin/,/bin/cp,/bin/ls,/bin/mv,/etc/,/etc/hosts, ...]

Great! Now let's see if we can write this function using an array comprehension using do notation.

Recall that a backwards arrow corresponds to choosing an element from an array. The first step is to choose an element from the immediate children of the argument. Then we simply call the function recursively for that file. Since we are using do notation, there is an implicit call to concatMap which concatenates all of the recursive results.

Here is the new version:

{{#include ../exercises/chapter4/test/Examples.purs:allFiles\_2}}

Try out the new version in PSCi - you should get the same result. I'll let you decide which version you find clearer.

## Exercises

1. (Easy) Write a function onlyFiles which returns all files (not directories) in all subdirectories of a directory.
2. (Medium) Write a function whereIs to search for a file by name. The function should return a value of type Maybe Path, indicating the directory containing the file, if it exists. It should behave as follows:
3. > whereIs root "ls"
4. Just (/bin/)
5. > whereIs root "cat"
6. Nothing

Hint: Try to write this function as an array comprehension using do notation.

1. (Difficult) Write a function largestSmallest which takes a Path and returns an array containing the single largest and single smallest files in the Path. Note: consider the cases where there are zero or one files in the Path by returning an empty array or a one-element array respectively.

## Conclusion

In this chapter, we covered the basics of recursion in PureScript, as a means of expressing algorithms concisely. We also introduced user-defined infix operators, standard functions on arrays such as maps, filters and folds, and array comprehensions which combine these ideas. Finally, we showed the importance of using tail recursion in order to avoid stack overflow errors, and how to use accumulator parameters to convert functions to tail recursive form.