PureScript by Example

This repository contains a community fork of *PureScript by Example* by Phil Freeman, also known as "the PureScript book". This version differs from the original in that it has been updated so that the code and exercises work with up-to-date versions of the compiler, libraries, and tools. Some chapters have also been rewritten to showcase the latest features of the PureScript ecosystem.

If you enjoyed the book or found it useful, please consider buying a copy of the original on Leanpub.

Translations: (Japanese)

Status

This book is being continuously updated as the language evolves, so please report any issues you discover with the material. We appreciate any feedback you have to share, even if it's as simple as pointing out a confusing section that we could make more beginner-friendly.

Unit tests are also being added to each chapter so you can check if your answers to the exercises are correct. See #79 for the latest status on tests.

About the Book

PureScript is a small, strongly, statically typed programming language with expressive types, written in and inspired by Haskell, and compiling to Javascript.

Functional programming in JavaScript has seen quite a lot of popularity recently, but large-scale application development is hindered by the lack of a disciplined environment in which to write code. PureScript aims to solve that problem by bringing the power of strongly-typed functional programming to the world of JavaScript development.

This book will show you how to get started with the PureScript programming language, from the basics (setting up a development environment) to the advanced.

Each chapter will be motivated by a particular problem, and in the course of solving that problem, new functional programming tools and techniques will be introduced. Here are

some examples of problems that will be solved in this book:

- Transforming data structures with maps and folds
- Form field validation using applicative functors
- Testing code with QuickCheck
- Using the canvas
- Domain specific language implementation
- Working with the DOM
- JavaScript interoperability
- Parallel asynchronous execution

License

Release

PureScript v0.15.15

Published on 2024-08-03

Introduction

Functional JavaScript

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

- Libraries such as UnderscoreJS 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 and virtual-dom model views as pure functions of application state.

Functions enable a simple form of abstraction that can yield great productivity gains.

However, functional programming in JavaScript has 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 that 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 that 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.

> Note that PureScript can target other backends, not only JavaScript, but this book focuses on targeting web browser and node environments.

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 that 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 that 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 that 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 the 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 is available from the book's GitHub repository. 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 is a great place to chat about issues you may be having. The server is dedicated to chatting about PureScript
- The PureScript Discourse Forum is another good place to search for solutions to common problems.
- PureScript: Jordan's 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 is a searchable database of PureScript types and functions. Read Pursuit's help page to learn what kinds of searches you can do.
- The unofficial PureScript Cookbook provides answers via code to "How do I do X?"-type questions.
- The PureScript documentation repository collects articles and examples on a wide variety of topics written by PureScript developers and users.
- The PureScript website contains links to several learning resources, including code samples, videos, and other resources for beginners.
- Try PureScript! is a website that 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, and can be reached on Twitter.

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.

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 helpful if that better suits your learning style.

Environment Setup

First, work through this Getting Started Guide 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 problem is confusing or contains unfamiliar syntax. We'll cover all of this in great detail in the upcoming chapters.

Editor support

You can use your preferred editor to write PureScript (for example, to solve the book exercises). See Editor Support Documentation.

> Note that some editors expect a `spago.dhall` file in the root of the opened project for full IDE support. For example, you should open the `chapter2` directory to work on the exercises in this chapter.
>
> If you use VS Code, you can use the provided workspace to open all the chapters simultaneously.

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 `prepareExercises.sh` script to simplify this process:

```
cd purescript-book
./scripts/prepareExercises.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 (located 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 a 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 `-}`:

```
    suite "diagonal" do
      test "3 4 5" do
        Assert.equal 5.0 (diagonal 3.0 4.0)
      test "5 12 13" do
        Assert.equal 13.0 (diagonal 5.0 12.0)
    {-  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 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:

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

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 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 helps with learning the material. If any of the exercises stumps you, please reach out to any of the community resources listed in the Getting Help section of this book, or even file an issue in this book's repo. 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. 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 only requires knowledge of the 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:

```
module Data.AddressBook where

import Prelude

import Control.Plus (empty)
import Data.List (List(..), filter, head)
import Data.Maybe (Maybe)
```

Here, we import several modules:

- The `Prelude` module, which contains a small set of standard definitions and functions. It re-exports many foundational modules from the `purescript-prelude` library.
- The `Control.Plus` module, which defines the `empty` value.
- The `Data.List` module, provided by the `lists` package, which can be installed using Spago. It contains a few functions that 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 (except for `Prelude`, which is typically imported as an open import). 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 corresponding 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 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 last example shows an error from the type checker, which failed 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 with the type `String` and an `interests` field with the type `Array String`, i.e., an array of `String`s.

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

```
import Prelude -- bring the (+) operator into scope

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

Alternatively, functions can be defined inline 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" using the `:paste` command. In this mode, declarations are terminated using the *Control-D* key sequence:

```
> import Prelude
> :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
```

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, any lines except the first must be indented past the indentation level of the first line.

Therefore, the following is a valid PureScript code:

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

But this is not a 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 introduce a new block of code, in which declarations must be further-indented:

```
example x y z =
  let
    foo = x * y
    bar = y * z
  in
    foo + bar
```

This doesn't compile:

```
example x y z =
  let
    foo = x * y
  bar = y * z
  in
    foo + bar
```

If you want to learn more (or encounter any problems), see the Syntax documentation.

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:

```
type Entry =
  { firstName :: String
  , lastName  :: String
  , address   :: Address
  }
```

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 the type `String`, and the `address` field will have the type `Address`, defined as follows:

```
type Address =
  { street :: String
  , city   :: String
  , state  :: String
  }
```

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:

```
type AddressBook = List Entry
```

Note that `List Entry` differs from `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.

Quantified Types

For illustration purposes, let's define a primitive function that takes any two arguments and returns the first one:

```
> :paste
… constantlyFirst :: forall a b. a -> b -> a
… constantlyFirst = \a b -> a
… ^D
```

> Note that if you use `:type` to ask about the type of `constantlyFirst`, it will be more verbose:

```
: type constantlyFirst
forall (a :: Type) (b :: Type). a -> b -> a
```

The type signature contains additional kind information, which explicitly notes that `a` and `b` should be concrete types.

The keyword `forall` indicates that `constantlyFirst` has a *universally quantified type*. It means we can substitute any types for `a` and `b` – `constantlyFirst` will work with these types.

For example, we might choose the type `a` to be `Int` and `b` to be `String`. In that case, we can *specialize* the type of `constantlyFirst` to

```
Int -> String -> Int
```

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

```
> constantlyFirst 3 "ignored"

3
```

While we can choose any types for `a` and `b`, the return type of `constantlyFirst` has to be the same as the type of the first argument (because both of them are "tied" to the same `a`):

```
:type constantlyFirst true "ignored"
Boolean

:type constantlyFirst "keep" 3
String
```

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:

```
showEntry :: Entry -> String
```

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

```
showEntry entry = entry.lastName <> ", " <>
                  entry.firstName <> ": " <>
                  showAddress entry.address
```

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:

```
showAddress :: Address -> String
showAddress addr = addr.street <> ", " <>
                   addr.city <> ", " <>
                   addr.state
```

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 representing an empty address book: an empty list.

```
emptyBook :: AddressBook
emptyBook = empty
```

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:

```
insertEntry :: Entry -> AddressBook -> AddressBook
```

This type signature says that `insertEntry` takes an `Entry` as its first argument, 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 :: Type). a -> List a -> List a
```

This type signature says that `Cons` takes a value of some type `a`, takes 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 an example of a *curried function*. In PureScript, all functions are considered curried.

Currying means converting a function that takes multiple arguments into a function that takes them one at a time. When we call a function, we pass it one argument, and it returns another function that also takes one argument until all arguments are passed.

For example, when we pass `5` to `add`, we get another function, which takes an int, adds 5 to it, and returns the sum as a result:

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

addFive :: Int -> Int
addFive = add 5
```

`addFive` is the result of *partial application*, which means we pass less than the total number of arguments to a function that takes multiple arguments. Let's give it a try:

Note that you must define the `add` function if you haven't already:

```
> import Prelude
> :paste
… add :: Int -> Int -> Int
… add x y = x + y
… ^D
```

```
> :paste
… addFive :: Int -> Int
… addFive = add 5
… ^D

> addFive 1
6

> add 5 1
6
```

To better understand currying and partial application, try making a few other functions, for example, out of `add`. And when you're done, let's return to the `insertEntry`.

```
insertEntry :: Entry -> AddressBook -> AddressBook
```

The `->` operator (in the type signature) associates to the right, which means that the compiler parses the type as

```
Entry -> (AddressBook -> AddressBook)
```

`insertEntry` 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 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, which explains why we can 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! 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:

```
insertEntry :: Entry -> AddressBook -> AddressBook
insertEntry = Cons
```

This process, called *eta conversion*, can be used (along with 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 the 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* 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 filter the address book, keeping only those entries with the correct first and last names. Then we can 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 :: Type). (a -> Boolean) -> List a -> List a

> :type head

forall (a :: Type). 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. 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:

```
findEntry :: String -> String -> AddressBook -> Maybe Entry
```

This type signature says that `findEntry` takes two strings: the first and last names, takes an `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 unnecessary to specify a type signature for `filterEntry`. However, doing so is recommended as a form of documentation.

Infix Function Application

Most 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, such as the `==` operator in the definition of `filterEntry`, where the operator is put *between* the two arguments. These infix operators are 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 ==
```

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.

In some situations, 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 of `5`, and make it right associative using `infixr`:

```
infixr 5 insertEntry as ++
```

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

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

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 a special syntax 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). 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. 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`:

```
findEntry firstName lastName = head <<< filter filterEntry
  where
    ...
```

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 decide which definition is easier to understand, but it is often useful to think of functions as building blocks in this way: each function executes a single task, and solutions are 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` that 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 the list head) is the one that is kept.

Conclusion

In this chapter, we covered several new functional programming concepts and learned how to:

- Use the interactive mode PSCi to experiment with functions and test ideas.
- Use types as both a correctness tool and an implementation tool.
- Use curried functions to represent functions of multiple arguments.
- Create programs from smaller components by composition.
- Structure code neatly using `where` expressions.
- Avoid null values by using the `Maybe` type.
- Use techniques like eta conversion and function composition to refactor code into a clear specification.

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

Pattern Matching

> Temporary note: If you're working on this chapter, beware that chapters 4 and 5 were swapped in November 2023.

Chapter Goals

This chapter will introduce two new concepts: algebraic data types and pattern matching. We will also briefly cover an interesting feature of the PureScript type system: row polymorphism.

Pattern matching is a common technique in functional programming and allows the developer to write compact functions, which express potentially complex ideas by breaking their implementation down into multiple cases.

Algebraic data types are a feature of the PureScript type system, which enables a similar level of expressiveness in the language of types – they are closely related to pattern matching.

The chapter's goal will be to write a library to describe and manipulate simple vector graphics using algebraic types and pattern matching.

Project Setup

The source code for this chapter is defined in the file `src/Data/Picture.purs`.

The `Data.Picture` module defines a data type `Shape` for simple shapes and a type `Picture` for collections of shapes, along with functions for working with those types.

The module imports the `Data.Foldable` module, which provides functions for folding data structures:

```
module Data.Picture where

import Prelude
import Data.Foldable (foldl)
import Data.Number (infinity)
```

The `Data.Picture` module also imports the `Number` module, but this time using the `as` keyword:

```
import Data.Number as Number
```

This makes the types and functions in that module available for use, but only by using the *qualified name*, like `Number.max`. This can be useful to avoid overlapping imports or clarify which modules certain things are imported from.

> *Note*: Using the same module name as the original module for a qualified import is unnecessary – shorter qualified names like `import Data.Number as N` are possible and quite common.

Simple Pattern Matching

Let's begin by looking at an example. Here is a function that computes the greatest common divisor of two integers using pattern matching:

```
gcd :: Int -> Int -> Int
gcd n 0 = n
gcd 0 m = m
gcd n m = if n > m
            then gcd (n - m) m
            else gcd n (m - n)
```

This algorithm is called the Euclidean Algorithm. If you search for its definition online, you will likely find a set of mathematical equations that look like the code above. One benefit of pattern matching is that it allows you to define code by cases, writing simple, declarative code that looks like a mathematical function specification.

A function written using pattern matching works by pairing sets of conditions with their results. Each line is called an *alternative* or a *case*. The expressions on the left of the equals sign are called *patterns*, and each case consists of one or more patterns separated by spaces. Cases describe which conditions the arguments must satisfy before the expression on the right of the equals sign should be evaluated and returned. Each case is tried in order, and the first case whose patterns match their inputs determines the return value.

For example, the `gcd` function is evaluated using the following steps:

- The first case is tried: if the second argument is zero, the function returns `n` (the first argument).
- If not, the second case is tried: if the first argument is zero, the function returns `m` (the second argument).
- Otherwise, the function evaluates and returns the expression in the last line.

Note that patterns can bind values to names – each line in the example binds one or both of the names `n` and `m` to the input values. As we learn about different patterns, we will see that different patterns correspond to different ways to choose names from the input arguments.

Simple Patterns

The example code above demonstrates two types of patterns:

- Integer literals patterns, which match something of type `Int`, only if the value matches exactly.
- Variable patterns, which bind their argument to a name

There are other types of simple patterns:

- `Number`, `String`, `Char`, and `Boolean` literals
- Wildcard patterns, indicated with an underscore (`_`), match any argument and do not bind any names.

Here are two more examples that demonstrate using these simple patterns:

```
fromString :: String -> Boolean
fromString "true" = true
fromString _      = false

toString :: Boolean -> String
toString true  = "true"
toString false = "false"
```

Try these functions in PSCi.

Guards

In the Euclidean algorithm example, we used an `if .. then .. else` expression to switch

between the two alternatives when `m > n` and `m <= n`. Another option, in this case, would be to use a *guard*.

A guard is a boolean-valued expression that must be satisfied in addition to the constraints imposed by the patterns. Here is the Euclidean algorithm rewritten to use a guard:

```
gcdV2 :: Int -> Int -> Int
gcdV2 n 0 = n
gcdV2 0 n = n
gcdV2 n m | n > m     = gcdV2 (n - m) m
          | otherwise = gcdV2 n (m - n)
```

In this case, the third line uses a guard to impose the extra condition that the first argument is strictly larger than the second. The guard in the final line uses the expression `otherwise`, which might seem like a keyword but is, in fact, just a regular binding in `Prelude`:

```
> :type otherwise
Boolean

> otherwise
true
```

This example demonstrates that guards appear on the left of the equals symbol, separated from the list of patterns by a pipe character (`|`).

Exercises

1. (Easy) Write the `factorial` function using pattern matching. *Hint*: Consider the two corner cases of zero and non-zero inputs. *Note*: This is a repeat of an example from the previous chapter, but see if you can rewrite it here on your own.
2. (Medium) Write a function `binomial` which finds the coefficient of the x^kth term in the polynomial expansion of $(1 + x)^n$. This is the same as the number of ways to choose a subset of k elements from a set of n elements. Use the formula $n! \ / \ k! \ (n - k)!$, where $!$ is the factorial function written earlier. *Hint*: Use pattern matching to handle corner cases. If it takes a long time to complete or crashes with an error about the call stack, try adding more corner cases.
3. (Medium) Write a function `pascal` which uses *Pascal`s Rule* for computing the same binomial coefficients as the previous exercise.

Array Patterns

Array literal patterns provide a way to match arrays of a fixed length. For example, suppose we want to write a function `isEmpty` which identifies empty arrays. We could do this by using an empty array pattern (`[]`) in the first alternative:

```
isEmpty :: forall a. Array a -> Boolean
isEmpty [] = true
isEmpty _ = false
```

Here is another function that matches arrays of length five, binding each of its five elements differently:

```
takeFive :: Array Int -> Int
takeFive [0, 1, a, b, _] = a * b
takeFive _ = 0
```

The first pattern only matches arrays with five elements, whose first and second elements are 0 and 1, respectively. In that case, the function returns the product of the third and fourth elements. In every other case, the function returns zero. For example, in PSCi:

```
> :paste
… takeFive [0, 1, a, b, _] = a * b
… takeFive _ = 0
… ^D

> takeFive [0, 1, 2, 3, 4]
6

> takeFive [1, 2, 3, 4, 5]
0

> takeFive []
0
```

Array literal patterns allow us to match arrays of a fixed length. Still, PureScript does *not* provide any means of matching arrays of an unspecified length since destructuring immutable arrays in these sorts of ways can lead to poor performance. If you need a data structure that supports this sort of matching, the recommended approach is to use `Data.List` . Other data structures exist which provide improved asymptotic performance for different operations.

Record Patterns and Row Polymorphism

Record patterns are used to match – you guessed it – records.

Record patterns look just like record literals, but instead of values on the right of the colon, we specify a binder for each field.

For example, this pattern matches any record which contains fields called `first` and `last`, and binds their values to the names `x` and `y`, respectively:

```
showPerson :: { first :: String, last :: String } -> String
showPerson { first: x, last: y } = y <> ", " <> x
```

Record patterns provide a good example of an interesting feature of the PureScript type system: *row polymorphism*. Suppose we had defined `showPerson` without a type signature above. What would its inferred type have been? Interestingly, it is not the same as the type we gave:

```
> showPerson { first: x, last: y } = y <> ", " <> x

> :type showPerson
forall (r :: Row Type). { first :: String, last :: String | r } -> String
```

What is the type variable `r` here? Well, if we try `showPerson` in PSCi, we see something interesting:

```
> showPerson { first: "Phil", last: "Freeman" }
"Freeman, Phil"

> showPerson { first: "Phil", last: "Freeman", location: "Los Angeles" }
"Freeman, Phil"
```

We can append additional fields to the record, and the `showPerson` function will still work. As long as the record contains the `first` and `last` fields of type `String`, the function application is well-typed. However, it is *not* valid to call `showPerson` with too *few* fields:

```
> showPerson { first: "Phil" }

Type of expression lacks required label "last"
```

We can read the new type signature of `showPerson` as "takes any record with `first` and `last` fields which are `Strings` *and any other fields*, and returns a `String`". This function is polymorphic in the *row* `r` of record fields, hence the name *row polymorphism*. Note that this

behavior is different than that of the original `showPerson`. Without the row variable `r`, `showPerson` only accepts records with *exactly* a `first` and `last` field and no others.

Note that we could have also written

```
> showPerson p = p.last <> ", " <> p.first
```

And PSCi would have inferred the same type.

Record Puns

Recall that the `showPerson` function matches a record inside its argument, binding the `first` and `last` fields to values named `x` and `y`. We could alternatively reuse the field names themselves and simplify this sort of pattern match as follows:

```
showPersonV2 :: { first :: String, last :: String } -> String
showPersonV2 { first, last } = last <> ", " <> first
```

Here, we only specify the names of the fields, and we do not need to specify the names of the values we want to introduce. This is called a *record pun*.

It is also possible to use record puns to *construct* records. For example, if we have values named `first` and `last` in scope, we can construct a person record using `{ first, last }`:

```
unknownPerson :: { first :: String, last :: String }
unknownPerson = { first, last }
  where
    first = "Jane"
    last  = "Doe"
```

This may improve the readability of code in some circumstances.

Nested Patterns

Array patterns and record patterns both combine smaller patterns to build larger patterns. For the most part, the examples above have only used simple patterns inside array patterns and record patterns. Still, it is important to note that patterns can be arbitrarily *nested*, which allows functions to be defined using conditions on potentially complex data types.

For example, this code combines two record patterns:

```
type Address = { street :: String, city :: String }

type Person = { name :: String, address :: Address }

livesInLA :: Person -> Boolean
livesInLA { address: { city: "Los Angeles" } } = true
livesInLA _ = false
```

Named Patterns

Patterns can be *named* to bring additional names into scope when using nested patterns. Any pattern can be named by using the `@` symbol.

For example, this function sorts two-element arrays, naming the two elements, but also naming the array itself:

```
sortPair :: Array Int -> Array Int
sortPair arr@[x, y]
  | x <= y = arr
  | otherwise = [y, x]
sortPair arr = arr
```

This way, we save ourselves from allocating a new array if the pair is already sorted. Note that if the input array does not contain *exactly* two elements, then this function returns it unchanged, even if it's unsorted.

Exercises

1. (Easy) Write a function `sameCity` which uses record patterns to test whether two `Person` records belong to the same city.
2. (Medium) What is the most general type of the `sameCity` function, considering row polymorphism? What about the `livesInLA` function defined above? *Note*: There is no test for this exercise.
3. (Medium) Write a function `fromSingleton` that uses an array literal pattern to extract the sole member of a singleton array. If the array is not a singleton, your function should return a provided default value. Your function should have type `forall a. a -> Array a -> a`

Case Expressions

Patterns do not only appear in top-level function declarations. It is possible to use patterns to match on an intermediate value in a computation using a `case` expression. Case expressions provide a similar type of utility to anonymous functions: it is not always desirable to give a name to a function, and a `case` expression allows us to avoid naming a function just because we want to use a pattern.

Here is an example. This function computes the "longest zero suffix" of an array (the longest suffix which sums to zero):

```
import Data.Array (tail)
import Data.Foldable (sum)
import Data.Maybe (fromMaybe)

lzs :: Array Int -> Array Int
lzs [] = []
lzs xs = case sum xs of
           0 -> xs
           _ -> lzs (fromMaybe [] $ tail xs)
```

For example:

```
> lzs [1, 2, 3, 4]
[]

> lzs [1, -1, -2, 3]
[-1, -2, 3]
```

This function works by case analysis. If the array is empty, our only option is to return an empty array. If the array is non-empty, we first use a `case` expression to split it into two cases. If the sum of the array is zero, we return the whole array. If not, we recurse on the tail of the array.

Pattern Match Failures and Partial Functions

If patterns in a case expression are tried in order, what happens when none of the patterns in a case alternatives match their inputs? In this case, the case expression will fail at runtime with a *pattern match failure*.

We can see this behavior with a simple example:

```
import Partial.Unsafe (unsafePartial)

partialFunction :: Boolean -> Boolean
partialFunction = unsafePartial \true -> true
```

This function contains only a single case, which only matches a single input, `true`. If we compile this file and test in PSCi with any other argument, we will see an error at runtime:

```
> partialFunction false

Failed pattern match
```

Functions that return a value for any combination of inputs are called *total* functions, and functions that do not are called *partial*.

It is generally considered better to define total functions where possible. If it is known that a function does not return a result for some valid set of inputs, it is usually better to return a value capable of indicating failure, such as type `Maybe a` for some `a`, using `Nothing` when it cannot return a valid result. This way, the presence or absence of a value can be indicated in a type-safe way.

The PureScript compiler will generate an error if it can detect that your function is not total due to an incomplete pattern match. The `unsafePartial` function can be used to silence these errors (if you are sure your partial function is safe!) If we removed the call to the `unsafePartial` function above, then the compiler would generate the following error:

```
A case expression could not be determined to cover all inputs.
The following additional cases are required to cover all inputs:

  false
```

This tells us that the value `false` is not matched by any pattern. In general, these warnings might include multiple unmatched cases.

If we also omit the type signature above:

```
partialFunction true = true
```

then PSCi infers a curious type:

```
> :type partialFunction

Partial => Boolean -> Boolean
```

We will see more types that involve the `=>` symbol later on in the book (they are related to *type classes*), but for now, it suffices to observe that PureScript keeps track of partial functions using the type system and that we must explicitly tell the type checker when they are safe.

The compiler will also generate a warning in certain cases when it can detect that cases are *redundant* (that is, a case only matches values which a prior case would have matched):

```
redundantCase :: Boolean -> Boolean
redundantCase true = true
redundantCase false = false
redundantCase false = false
```

In this case, the last case is correctly identified as redundant:

```
A case expression contains unreachable cases:

  false
```

Note: PSCi does not show warnings, so to reproduce this example, you will need to save this function as a file and compile it using `spago build`.

Algebraic Data Types

This section will introduce a feature of the PureScript type system called *Algebraic Data Types* (or *ADTs*), which are fundamentally related to pattern matching.

However, we'll first consider a motivating example, which will provide the basis of a solution to this chapter's problem of implementing a simple vector graphics library.

Suppose we wanted to define a type to represent some simple shapes: lines, rectangles, circles, text, etc. In an object oriented language, we would probably define an interface or abstract class `Shape`, and one concrete subclass for each type of shape that we wanted to be able to work with.

However, this approach has one major drawback: to work with `Shape`s abstractly, it is necessary to identify all of the operations one might wish to perform and to define them on the `Shape` interface. It becomes difficult to add new operations without breaking modularity.

Algebraic data types provide a type-safe way to solve this problem if the set of shapes is known in advance. It is possible to define new operations on `Shape` in a modular way and still maintain type-safety.

Here is how `Shape` might be represented as an algebraic data type:

```
data Shape
  = Circle Point Number
  | Rectangle Point Number Number
  | Line Point Point
  | Text Point String

type Point =
  { x :: Number
  , y :: Number
  }
```

This declaration defines `Shape` as a sum of different constructors, and for each constructor identifies the included data. A `Shape` is either a `Circle` that contains a center `Point` and a radius (a number), or a `Rectangle`, or a `Line`, or `Text`. There are no other ways to construct a value of type `Shape`.

An algebraic data type is introduced using the `data` keyword, followed by the name of the new type and any type arguments. The type's constructors (i.e., its *data constructors*) are defined after the equals symbol and separated by pipe characters (`|`). The data carried by an ADT's constructors doesn't have to be restricted to primitive types: constructors can include records, arrays, or even other ADTs.

Let's see another example from PureScript's standard libraries. We saw the `Maybe` type, which is used to define optional values, earlier in the book. Here is its definition from the `maybe` package:

```
data Maybe a = Nothing | Just a
```

This example demonstrates the use of a type parameter `a`. Reading the pipe character as the word "or", its definition almost reads like English: "a value of type `Maybe a` is either `Nothing`, or `Just` a value of type `a`".

Note that we don't use the syntax `forall a.` anywhere in our data definition. `forall` syntax is necessary for functions but is not used when defining ADTs with `data` or type aliases with `type`.

Data constructors can also be used to define recursive data structures. Here is one more example, defining a data type of singly-linked lists of elements of type `a`:

```
data List a = Nil | Cons a (List a)
```

This example is taken from the `lists` package. Here, the `Nil` constructor represents an empty list, and `Cons` is used to create non-empty lists from a head element and a tail. Notice how the tail is defined using the data type `List a`, making this a recursive data type.

Using ADTs

It is simple enough to use the constructors of an algebraic data type to construct a value: simply apply them like functions, providing arguments corresponding to the data included with the appropriate constructor.

For example, the `Line` constructor defined above required two `Point`s, so to construct a `Shape` using the `Line` constructor, we have to provide two arguments of type `Point`:

```
exampleLine :: Shape
exampleLine = Line p1 p2
  where
    p1 :: Point
    p1 = { x: 0.0, y: 0.0 }

    p2 :: Point
    p2 = { x: 100.0, y: 50.0 }
```

So, constructing values of algebraic data types is simple, but how do we use them? This is where the important connection with pattern matching appears: the only way to consume a value of an algebraic data type is to use a pattern to match its constructor.

Let's see an example. Suppose we want to convert a `Shape` into a `String`. We have to use pattern matching to discover which constructor was used to construct the `Shape`. We can do this as follows:

```
showShape :: Shape -> String
showShape (Circle c r) =
  "Circle [center: " <> showPoint c <> ", radius: " <> show r <> "]"
showShape (Rectangle c w h) =
  "Rectangle [center: " <> showPoint c <> ", width: " <> show w <> ", height: "
<> show h <> "]"
showShape (Line start end) =
  "Line [start: " <> showPoint start <> ", end: " <> showPoint end <> "]"
showShape (Text loc text) =
  "Text [location: " <> showPoint loc <> ", text: " <> show text <> "]"

showPoint :: Point -> String
showPoint { x, y } =
  "(" <> show x <> ", " <> show y <> ")"
```

Each constructor can be used as a pattern, and the arguments to the constructor can themselves be bound using patterns of their own. Consider the first case of `showShape`: if the `Shape` matches the `Circle` constructor, then we bring the arguments of `Circle` (center and radius) into scope using two variable patterns, `c` and `r`. The other cases are similar.

Exercises

1. (Easy) Write a function `circleAtOrigin` which constructs a `Circle` (of type `Shape`) centered at the origin with a radius `10.0`.
2. (Medium) Write a function `doubleScaleAndCenter` that scales the size of a `Shape` by a factor of `2.0` and centers it at the origin.
3. (Medium) Write a function `shapeText` which extracts the text from a `Shape`. It should return `Maybe String`, and use the `Nothing` constructor if the input is not constructed using `Text`.

Newtypes

There is a special case of algebraic data types, called *newtypes*. Newtypes are introduced using the `newtype` keyword instead of the `data` keyword.

Newtypes must define *exactly one* constructor, and that constructor must take *exactly one* argument. That is, a newtype gives a new name to an existing type. In fact, the values of a newtype have the same runtime representation as the underlying type, so there is no runtime performance overhead. They are, however, distinct from the point of view of the

type system. This gives an extra layer of type safety.

As an example, we might want to define newtypes as type-level aliases for `Number`, to ascribe units like volts, amps, and ohms:

```
newtype Volt = Volt Number
newtype Ohm = Ohm Number
newtype Amp = Amp Number
```

Then we define functions and values using these types:

```
calculateCurrent :: Volt -> Ohm -> Amp
calculateCurrent (Volt v) (Ohm r) = Amp (v / r)

battery :: Volt
battery = Volt 1.5

lightbulb :: Ohm
lightbulb = Ohm 500.0

current :: Amp
current = calculateCurrent battery lightbulb
```

This prevents us from making silly mistakes, such as attempting to calculate the current produced by *two* lightbulbs *without* a voltage source.

```
current :: Amp
current = calculateCurrent lightbulb lightbulb
{-
TypesDoNotUnify:
  current = calculateCurrent lightbulb lightbulb
                             ^^^^^^^^^
  Could not match type
    Ohm
  with type
    Volt
-}
```

If we instead just used `Number` without `newtype`, then the compiler can't help us catch this mistake:

```
-- This also compiles, but is not as type safe.
calculateCurrent :: Number -> Number -> Number
calculateCurrent v r = v / r

battery :: Number
battery = 1.5

lightbulb :: Number
lightbulb = 500.0

current :: Number
current = calculateCurrent lightbulb lightbulb -- uncaught mistake
```

Note that while a newtype can only have a single constructor, and the constructor must be of a single value, a newtype *can* take any number of type variables. For example, the following newtype would be a valid definition (`err` and `a` are the type variables, and the `CouldError` constructor expects a *single* value of type `Either err a`):

```
newtype CouldError err a = CouldError (Either err a)
```

Also, note that the constructor of a newtype often has the same name as the newtype itself, but this is not a requirement. For example, unique names are also valid:

```
newtype Coulomb = MakeCoulomb Number
```

In this case, `Coulomb` is the *type constructor* (of zero arguments), and `MakeCoulomb` is the *data constructor*. These constructors live in different namespaces, even when the names are identical, such as with the `Volt` example. This is true for all ADTs. Note that although the type constructor and data constructor can have different names, in practice, it is idiomatic for them to share the same name. This is the case with `Amp` and `Volt` types above.

Another application of newtypes is to attach different *behavior* to an existing type without changing its representation at runtime. We cover that use case in the next chapter when we discuss *type classes*.

Exercises

1. (Easy) Define `Watt` as a `newtype` of `Number` . Then define a `calculateWattage` function using this new `Watt` type and the above definitions `Amp` and `Volt` :

```
calculateWattage :: Amp -> Volt -> Watt
```

A wattage in `Watt` s can be calculated as the product of a given current in `Amp` s and a given voltage in `Volt` s.

A Library for Vector Graphics

Let's use the data types we have defined above to create a simple library for using vector graphics.

Define a type synonym for a `Picture` – just an array of `Shape` s:

```
type Picture = Array Shape
```

For debugging purposes, we'll want to be able to turn a `Picture` into something readable. The `showPicture` function lets us do that:

```
showPicture :: Picture -> Array String
showPicture = map showShape
```

Let's try it out. Compile your module with `spago build` and open PSCi with `spago repl` :

```
$ spago build
$ spago repl

> import Data.Picture

> showPicture [ Line { x: 0.0, y: 0.0 } { x: 1.0, y: 1.0 } ]

["Line [start: (0.0, 0.0), end: (1.0, 1.0)]"]
```

Computing Bounding Rectangles

The example code for this module contains a function `bounds` which computes the smallest bounding rectangle for a `Picture` .

The `Bounds` type defines a bounding rectangle.

```
type Bounds =
  { top    :: Number
  , left   :: Number
  , bottom :: Number
  , right  :: Number
  }
```

`bounds` uses the `foldl` function from `Data.Foldable` to traverse the array of `Shapes` in a `Picture`, and accumulate the smallest bounding rectangle:

```
bounds :: Picture -> Bounds
bounds = foldl combine emptyBounds
  where
  combine :: Bounds -> Shape -> Bounds
  combine b shape = union (shapeBounds shape) b
```

In the base case, we need to find the smallest bounding rectangle of an empty `Picture`, and the empty bounding rectangle defined by `emptyBounds` suffices.

The accumulating function `combine` is defined in a `where` block. `combine` takes a bounding rectangle computed from `foldl`'s recursive call, and the next `Shape` in the array, and uses the `union` function to compute the union of the two bounding rectangles. The `shapeBounds` function computes the bounds of a single shape using pattern matching.

Exercises

1. (Medium) Extend the vector graphics library with a new operation `area` that computes the area of a `Shape`. For the purpose of this exercise, the area of a line or a piece of text is assumed to be zero.
2. (Difficult) Extend the `Shape` type with a new data constructor `Clipped`, which clips another `Picture` to a rectangle. Extend the `shapeBounds` function to compute the bounds of a clipped picture. Note that this makes `Shape` into a recursive data type. *Hint*: The compiler will walk you through extending other functions as required.

Conclusion

In this chapter, we covered pattern matching, a basic but powerful technique from functional programming. We saw how to use simple patterns as well as array and record patterns to match parts of deep data structures.

This chapter also introduced algebraic data types, which are closely related to pattern matching. We saw how algebraic data types allow concise descriptions of data structures and provide a modular way to extend data types with new operations.

Finally, we covered *row polymorphism*, a powerful type of abstraction that allows many idiomatic JavaScript functions to be given a type.

In the rest of the book, we will use ADTs and pattern matching extensively, so it will pay dividends to become familiar with them now. Try creating your own algebraic data types and writing functions to consume them using pattern matching.

Recursion, Maps, And Folds

> Temporary note: If you're working on this chapter, beware that chapters 4 and 5 were swapped in November 2023.

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 `map`, `fold`, and 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 that 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:

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

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

Here is another common example that computes the *Fibonacci function*:

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

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.

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)

length :: forall a. Array a -> Int
length [] = 0
length arr = 1 + (length $ fromMaybe [] $ tail arr)
```

In this function, we branch based on the emptiness of the array. The `null` function returns `true` on an empty array. Empty arrays have a length of 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 a very impractical way to find the length of an array in JavaScript, but it should provide enough help to allow you to complete the following exercises:

Exercises

1. (Easy) Write a recursive function `isEven` that returns `true` if and only if its input is an even integer.
2. (Medium) Write a recursive function `countEven` that 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 that 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 `<$>`.

```
> (\n -> n + 1) <$> [1, 2, 3, 4, 5]
[2, 3, 4, 5, 6]
```

Let's look at the type of `map`:

```
> :type map
forall (f :: Type -> Type) (a :: Type) (b :: Type). 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 :: Type) (b :: Type). (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 `<$>` looks like special syntax, it is in fact just an alias for a regular PureScript function. The function is simply *applied* using infix syntax. In fact, the function can be used like a regular function by enclosing its name in parentheses. This means that we can use the parenthesized name `(<$>)` 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 :: Type). 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 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 :: Type) (b :: Type). (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 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:

```
factors :: Int -> Array (Array Int)
factors n = filter (\xs -> product xs == n) do
  i <- 1 .. n
  j <- i .. n
  pure [ i, j ]
```

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

- Expressions that 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 that 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 that 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. We can modify our `factors` function to use this form, instead of using `pure`:

```
factorsV2 :: Int -> Array (Array Int)
factorsV2 n = filter (\xs -> product xs == n) do
  i <- 1 .. n
  j <- i .. n
  [ [ i, j ] ]
```

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)

factorsV3 :: Int -> Array (Array Int)
factorsV3 n = do
  i <- 1 .. n
  j <- i .. n
  guard $ i * j == n
  pure [ i, j ]
```

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 :: Type -> Type). Alternative m => Boolean -> m Unit
```

The `Unit` type represents values with no computational content — the absence of a concrete meaningful value.

We often use `Unit` "wrapped" in a type constructor as the return type of a computation where we only care about the *effects* of the computation (or a "shape" of the result) and not some concrete value.

For example, the `main` function has the type `Effect Unit`. Main is an entry point to the project — we don't call it directly.

We'll explain what `m` in the type signature means in Chapter 6.

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
```

If we pass an expression to `guard` that 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 whether its integer argument is prime. *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 $a^2 + b^2 = c^2$. *Hint*: Use the `guard` function in an array comprehension.
4. (Difficult) Write a function `primeFactors` which produces the prime factorization of `n`, i.e., the array of prime integers whose product is `n`. *Hint*: for an integer greater than 1, break the problem into two subproblems: finding the first factor and the remaining factors.

Folds

Left and right folds over arrays provide another class of interesting functions that 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 (f :: Type -> Type) (a :: Type) (b :: Type). Foldable f => (b -> a -> b)
-> b -> f a -> b

> :type foldr
forall (f :: Type -> Type) (a :: Type) (b :: Type). Foldable f => (a -> b -> b)
-> b -> f a -> b
```

These types are more general than we are interested in right now. For this chapter, we can simplify and assume the following (more specific) type signatures:

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

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

In both 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 `Int`s to add. For the first argument, we can 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 matters 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 adopt recursion as a standard technique from functional programming, we need a way to deal with possibly unbounded recursion.

PureScript provides a partial solution to this problem through *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` and `tailrec` packages.

The key observation that enables tail recursion optimization: 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 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:

```
factorialTailRec :: Int -> Int -> Int
factorialTailRec 0 acc = acc
factorialTailRec n acc = factorialTailRec (n - 1) (acc * n)
```

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

Accumulators

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

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

```
length :: forall a. Array a -> Int
length [] = 0
length arr = 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:

```
lengthTailRec :: forall a. Array a -> Int
lengthTailRec arr = length' arr 0
  where
  length' :: Array a -> Int -> Int
  length' [] acc = acc
  length' arr' acc = length' (fromMaybe [] $ tail arr') (acc + 1)
```

In this case, we delegate to the helper function `length'`, which is tail recursive – its only recursive call is in the last case, in tail position. This means that the generated code will be a *while loop* and 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 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 a "state", there is no direct mutation.

Prefer Folds to Explicit Recursion

If we can write our recursive functions using tail recursion, 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'll apply what we've learned, writing functions that 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` representing 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 that 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 that performs a deep enumeration of all files inside a directory. This function will have the following type:

```
allFiles :: Path -> Array Path
```

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 simultaneously.

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

```
allFiles file = file : concatMap allFiles (ls file)
```

> *Note*: the cons operator `:` has poor performance on immutable arrays, so it is not generally recommended. 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 call the function recursively for that file. Since we use do notation, there is an implicit call to `concatMap`, which concatenates all of the recursive results.

Here is the new version:

```
allFiles' :: Path -> Array Path
allFiles' file = file : do
  child <- ls file
  allFiles' child
```

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:

   ```
   > whereIs root "ls"
   Just (/bin/)

   > whereIs root "cat"
   Nothing
   ```

   Hint: Try to write this function as an array comprehension using do notation.
3. (Difficult) Write a function `largestSmallest` which takes a `Path` and returns an array containing the single largest and single smallest files in the `Path`, including (recursively) any subdirectories. *Note*: consider the cases where there are zero or one files in the `Path` by returning an empty or one-element array, respectively.

Conclusion

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

Type Classes

Chapter Goals

This chapter will introduce a powerful form of abstraction enabled by PureScript's type system – type classes.

This motivating example for this chapter will be a library for hashing data structures. We will see how the machinery of type classes allows us to hash complex data structures without having to think directly about the structure of the data itself.

We will also see a collection of standard type classes from PureScript's Prelude and standard libraries. PureScript code leans heavily on the power of type classes to express ideas concisely, so it will be beneficial to familiarize yourself with these classes.

If you come from an Object Oriented background, please note that the word "class" means something *very* different in this context than what you're used to. A type class serves a purpose more similar to an OO interface.

Project Setup

The source code for this chapter is defined in the file `src/Data/Hashable.purs`.

The project has the following dependencies:

- `maybe`, which defines the `Maybe` data type, which represents optional values.
- `tuples`, which defines the `Tuple` data type, which represents pairs of values.
- `either`, which defines the `Either` data type, which represents disjoint unions.
- `strings`, which defines functions that operate on strings.
- `functions`, which defines some helper functions for defining PureScript functions.

The module `Data.Hashable` imports several modules provided by these packages.

Show Me!

Our first simple example of a type class is provided by a function we've seen several times

already: the `show` function, which takes a value and displays it as a string.

`show` is defined by a type class in the `Prelude` module called `Show`, which is defined as follows:

```
class Show a where
  show :: a -> String
```

This code declares a new *type class* called `Show`, which is parameterized by the type variable `a`.

A type class *instance* contains implementations of the functions defined in a type class, specialized to a particular type.

For example, here is the definition of the `Show` type class instance for `Boolean` values, taken from the Prelude:

```
instance Show Boolean where
  show true = "true"
  show false = "false"
```

This code declares a type class instance; we say that the `Boolean` type *belongs to the `Show` type class*.

If you're wondering, the generated JS code looks like this:

```
var showBoolean = {
    show: function (v) {
        if (v) {
            return "true";
        };
       if (!v) {
            return "false";
        };
        throw new Error("Failed pattern match at ...");
    }
};
```

If you're unhappy with the generated name, you can give names to type class instances. For example:

```
instance myShowBoolean :: Show Boolean where
  show true = "true"
  show false = "false"
```

```
var myShowBoolean = {
    show: function (v) {
        if (v) {
            return "true";
        };
       if (!v) {
            return "false";
        };
        throw new Error("Failed pattern match at ...");
    }
};
```

We can try out the `Show` type class in PSCi by showing a few values with different types:

```
> import Prelude

> show true
"true"

> show 1.0
"1.0"

> show "Hello World"
"\"Hello World\""
```

These examples demonstrate how to `show` values of various primitive types, but we can also `show` values with more complicated types:

```
> import Data.Tuple

> show (Tuple 1 true)
"(Tuple 1 true)"

> import Data.Maybe

> show (Just "testing")
"(Just \"testing\")"
```

The output of `show` should be a string that you can paste back into the repl (or `.purs` file) to recreate the item being shown. Here we'll use `logShow`, which just calls `show` and then `log`, to render the string without quotes. Ignore the `unit` print – that will be covered in Chapter 8 when we examine `Effect`s, like `log`.

```
> import Effect.Console

> logShow (Tuple 1 true)
(Tuple 1 true)
unit

> logShow (Just "testing")
(Just "testing")
unit
```

If we try to show a value of type `Data.Either`, we get an interesting error message:

```
> import Data.Either
> show (Left 10)

The inferred type

    forall a. Show a => String

has type variables which are not mentioned in the body of the type. Consider
adding a type annotation.
```

The problem here is not that there is no `Show` instance for the type we intended to `show`, but rather that PSCi could not infer the type. This is indicated by the *unknown type* `a` in the inferred type.

We can annotate the expression with a type using the `::` operator, so that PSCi can choose the correct type class instance:

```
> show (Left 10 :: Either Int String)
"(Left 10)"
```

Some types do not have a `Show` instance defined at all. One example of this is the function type `->`. If we try to `show` a function from `Int` to `Int`, we get an appropriate error message from the type checker:

```
> import Prelude
> show $ \n -> n + 1

No type class instance was found for

  Data.Show.Show (Int -> Int)
```

Type class instances can be defined in one of two places: in the same module that the type class is defined, or in the same module that the type "belonging to" the type class is defined.

An instance defined in any other spot is called an "orphan instance" and is not allowed by the PureScript compiler. Some of the exercises in this chapter will require you to copy the definition of a type into your MySolutions module so that you can define type class instances for that type.

Exercises

1. (Easy) Define a `Show` instance for `Point`. Match the same output as the `showPoint` function from the previous chapter. *Note:* Point is now a `newtype` (instead of a `type` synonym), which allows us to customize how to `show` it. Otherwise, we'd be stuck with the default `Show` instance for records.

```
newtype Point
  = Point
  { x :: Number
  , y :: Number
  }
```

Common Type Classes

In this section, we'll look at some standard type classes defined in the Prelude and standard libraries. These type classes form the basis of many common patterns of abstraction in idiomatic PureScript code, so a basic understanding of their functions is highly recommended.

Eq

The `Eq` type class defines the `eq` function, which tests two values for equality. The `==` operator is actually an alias for `eq`.

```
class Eq a where
  eq :: a -> a -> Boolean
```

In either case, the two arguments must have the same type: it does not make sense to compare two values of different types for equality.

Try out the `Eq` type class in PSCi:

```
> 1 == 2
false

> "Test" == "Test"
true
```

Ord

The `Ord` type class defines the `compare` function, which can be used to compare two values, for types that support ordering. The comparison operators `<` and `>` along with their non-strict companions `<=` and `>=`, can be defined in terms of `compare`.

Note: In the example below, the class signature contains `<=`. This usage of `<=` in this context indicates that `Eq` is a superclass of `Ord` and is not intended to represent the use of `<=` as a comparison operator. See the section Superclasses below.

```
data Ordering = LT | EQ | GT

class Eq a <= Ord a where
  compare :: a -> a -> Ordering
```

The `compare` function compares two values and returns an `Ordering`, which has three alternatives:

- `LT` – if the first argument is less than the second.
- `EQ` – if the first argument is equal to the second.
- `GT` – if the first argument is greater than the second.

Again, we can try out the `compare` function in PSCi:

```
> compare 1 2
LT

> compare "A" "Z"
LT
```

Field

The `Field` type class identifies those types which support numeric operators such as

addition, subtraction, multiplication, and division. It is provided to abstract over those operators, so that they can be reused where appropriate.

> *Note*: Just like the `Eq` and `Ord` type classes, the `Field` type class has special support in the PureScript compiler, so that simple expressions such as `1 + 2 * 3` get translated into simple JavaScript, as opposed to function calls which dispatch based on a type class implementation.

```
class EuclideanRing a <= Field a
```

The `Field` type class is composed from several more general *superclasses*. This allows us to talk abstractly about types that support some but not all of the `Field` operations. For example, a type of natural numbers would be closed under addition and multiplication, but not necessarily under subtraction, so that type might have an instance of the `Semiring` class (which is a superclass of `Num`), but not an instance of `Ring` or `Field`.

Superclasses will be explained later in this chapter, but the full numeric type class hierarchy (cheatsheet) is beyond the scope of this chapter. The interested reader is encouraged to read the documentation for the superclasses of `Field` in `prelude`.

Semigroups and Monoids

The `Semigroup` type class identifies those types which support an `append` operation to combine two values:

```
class Semigroup a where
  append :: a -> a -> a
```

Strings form a semigroup under regular string concatenation, and so do arrays. The `prelude` package provides several other standard instances.

The `<>` concatenation operator, which we have already seen, is provided as an alias for `append`.

The `Monoid` type class (provided by the `prelude` package) extends the `Semigroup` type class with the concept of an empty value, called `mempty`:

```
class Semigroup m <= Monoid m where
  mempty :: m
```

Again, strings and arrays are simple examples of monoids.

A `Monoid` type class instance for a type describes how to *accumulate* a result with that type by starting with an "empty" value and combining new results. For example, we can write a function that concatenates an array of values in some monoid using a fold. In PSCi:

```
> import Prelude
> import Data.Monoid
> import Data.Foldable

> foldl append mempty ["Hello", " ", "World"]
"Hello World"

> foldl append mempty [[1, 2, 3], [4, 5], [6]]
[1,2,3,4,5,6]
```

The `prelude` package provides many examples of monoids and semigroups, which we will use in the rest of the book.

Foldable

If the `Monoid` type class identifies those types which act as the result of a fold, then the `Foldable` type class identifies those type constructors which can be used as the source of a fold.

The `Foldable` type class is provided in the `foldable-traversable` package, which also contains instances for some standard containers such as arrays and `Maybe`.

The type signatures for the functions belonging to the `Foldable` class are a little more complicated than the ones we've seen so far:

```
class Foldable f where
  foldr :: forall a b. (a -> b -> b) -> b -> f a -> b
  foldl :: forall a b. (b -> a -> b) -> b -> f a -> b
  foldMap :: forall a m. Monoid m => (a -> m) -> f a -> m
```

It is instructive to specialize to the case where `f` is the array type constructor. In this case, we can replace `f a` with `Array a` for any a, and we notice that the types of `foldl` and `foldr` become the types we saw when we first encountered folds over arrays.

What about `foldMap`? Well, that becomes `forall a m. Monoid m => (a -> m) -> Array a -> m`. This type signature says that we can choose any type `m` for our result type, as long as that type is an instance of the `Monoid` type class. If we can provide a function that turns our

array elements into values in that monoid, then we can accumulate over our array using the structure of the monoid and return a single value.

Let's try out `foldMap` in PSCi:

```
> import Data.Foldable

> foldMap show [1, 2, 3, 4, 5]
"12345"
```

Here, we choose the monoid for strings, which concatenates strings together, and the `show` function, which renders an `Int` as a `String`. Then, passing in an array of integers, we see that the results of `show`ing each integer have been concatenated into a single `String`.

But arrays are not the only types that are foldable. `foldable-traversable` also defines `Foldable` instances for types like `Maybe` and `Tuple`, and other libraries like `lists` define `Foldable` instances for their own data types. `Foldable` captures the notion of an *ordered container*.

Functor and Type Class Laws

The Prelude also defines a collection of type classes that enable a functional style of programming with side-effects in PureScript: `Functor`, `Applicative`, and `Monad`. We will cover these abstractions later in the book, but for now, let's look at the definition of the `Functor` type class, which we have seen already in the form of the `map` function:

```
class Functor f where
  map :: forall a b. (a -> b) -> f a -> f b
```

The `map` function (and its alias `<$>`) allows a function to be "lifted" over a data structure. The precise definition of the word "lifted" here depends on the data structure in question, but we have already seen its behavior for some simple types:

```
> import Prelude

> map (\n -> n < 3) [1, 2, 3, 4, 5]
[true, true, false, false, false]

> import Data.Maybe
> import Data.String (length)

> map length (Just "testing")
(Just 7)
```

How can we understand the meaning of the `map` function, when it acts on many different structures, each in a different way?

Well, we can build an intuition that the `map` function applies the function it is given to each element of a container, and builds a new container from the results, with the same shape as the original. But how do we make this concept precise?

Type class instances for `Functor` are expected to adhere to a set of *laws*, called the *functor laws*:

- `map identity xs = xs`
- `map g (map f xs) = map (g <<< f) xs`

The first law is the *identity law*. It states that lifting the identity function (the function which returns its argument unchanged) over a structure just returns the original structure. This makes sense since the identity function does not modify its input.

The second law is the *composition law*. It states that mapping one function over a structure and then mapping a second is the same as mapping the composition of the two functions over the structure.

Whatever "lifting" means in the general sense, it should be true that any reasonable definition of lifting a function over a data structure should obey these rules.

Many standard type classes come with their own set of similar laws. The laws given to a type class give structure to the functions of that type class and allow us to study its instances in generality. The interested reader can research the laws ascribed to the standard type classes that we have seen already.

Deriving Instances

Rather than writing instances manually, you can let the compiler do most of the work for you. Take a look at this Type Class Deriving guide. That information will help you solve the following exercises.

Exercises

The following newtype represents a complex number:

```
newtype Complex
  = Complex
  { real :: Number
  , imaginary :: Number
  }
```

1. (Easy) Define a `Show` instance for `Complex`. Match the output format expected by the tests (e.g. `1.2+3.4i`, `5.6-7.8i`, etc.).

2. (Easy) Derive an `Eq` instance for `Complex`. *Note*: You may instead write this instance manually, but why do more work if you don't have to?

3. (Medium) Define a `Semiring` instance for `Complex`. *Note*: You can use `wrap` and `over2` from `Data.Newtype` to create a more concise solution. If you do so, you will also need to import `class Newtype` from `Data.Newtype` and derive a `Newtype` instance for `Complex`.

4. (Easy) Derive (via `newtype`) a `Ring` instance for `Complex`. *Note*: You may instead write this instance manually, but that's not as convenient.

   Here's the `Shape` ADT from the previous chapter:

   ```
   data Shape
     = Circle Point Number
     | Rectangle Point Number Number
     | Line Point Point
     | Text Point String
   ```

5. (Medium) Derive (via `Generic`) a `Show` instance for `Shape`. How does the amount of code written and `String` output compare to `showShape` from the previous chapter? *Hint*: See the Deriving from `Generic` section of the Type Class Deriving guide.

Type Class Constraints

Types of functions can be constrained by using type classes. Here is an example: suppose we want to write a function that tests if three values are equal, by using equality defined using an `Eq` type class instance.

```
threeAreEqual :: forall a. Eq a => a -> a -> a -> Boolean
threeAreEqual a1 a2 a3 = a1 == a2 && a2 == a3
```

The type declaration looks like an ordinary polymorphic type defined using `forall`. However, there is a type class constraint `Eq a`, separated from the rest of the type by a double arrow `=>`.

This type says that we can call `threeAreEqual` with any choice of type `a`, as long as there is an `Eq` instance available for `a` in one of the imported modules.

Constrained types can contain several type class instances, and the types of the instances are not restricted to simple type variables. Here is another example which uses `Ord` and `Show` instances to compare two values:

```
showCompare :: forall a. Ord a => Show a => a -> a -> String
showCompare a1 a2 | a1 < a2 =
  show a1 <> " is less than " <> show a2
showCompare a1 a2 | a1 > a2 =
  show a1 <> " is greater than " <> show a2
showCompare a1 a2 =
  show a1 <> " is equal to " <> show a2
```

Note that multiple constraints can be specified by using the `=>` symbol multiple times, just like we specify curried functions of multiple arguments. But remember not to confuse the two symbols:

- `a -> b` denotes the type of functions from *type* `a` to *type* `b`, whereas
- `a => b` applies the *constraint* `a` to the type `b`.

The PureScript compiler will try to infer constrained types when a type annotation is not provided. This can be useful if we want to use the most general type possible for a function.

To see this, try using one of the standard type classes like `Semiring` in PSCi:

```
> import Prelude

> :type \x -> x + x
forall (a :: Type). Semiring a => a -> a
```

Here, we might have annotated this function as `Int -> Int` or `Number -> Number`, but PSCi shows us that the most general type works for any `Semiring`, allowing us to use our function with both `Int`s and `Number`.

Instance Dependencies

Just as the implementation of functions can depend on type class instances using constrained types, so can the implementation of type class instances depend on other type class instances. This provides a powerful form of program inference, in which the implementation of a program can be inferred using its types.

For example, consider the `Show` type class. We can write a type class instance to `show` arrays of elements, as long as we have a way to `show` the elements themselves:

```
instance Show a => Show (Array a) where
  ...
```

If a type class instance depends on multiple other instances, those instances should be grouped in parentheses and separated by commas on the left-hand side of the `=>` symbol:

```
instance (Show a, Show b) => Show (Either a b) where
  ...
```

These two type class instances are provided in the `prelude` library.

When the program is compiled, the correct type class instance for `Show` is chosen based on the inferred type of the argument to `show`. The selected instance might depend on many such instance relationships, but this complexity is not exposed to the developer.

Exercises

1. (Easy) The following declaration defines a type of non-empty arrays of elements of type `a`:

   ```
   data NonEmpty a = NonEmpty a (Array a)
   ```

   Write an `Eq` instance for the type `NonEmpty a` that reuses the instances for `Eq a` and `Eq (Array a)`. *Note:* you may instead derive the `Eq` instance.

2. (Medium) Write a `Semigroup` instance for `NonEmpty a` by reusing the `Semigroup` instance for `Array`.

3. (Medium) Write a `Functor` instance for `NonEmpty`.

4. (Medium) Given any type `a` with an instance of `Ord`, we can add a new "infinite" value that is greater than any other value:

```
data Extended a = Infinite | Finite a
```

Write an `Ord` instance for `Extended a` that reuses the `Ord` instance for `a`.

5. (Difficult) Write a `Foldable` instance for `NonEmpty`. *Hint*: reuse the `Foldable` instance for arrays.

6. (Difficult) Given a type constructor `f` which defines an ordered container (and so has a `Foldable` instance), we can create a new container type that includes an extra element at the front:

   ```
   data OneMore f a = OneMore a (f a)
   ```

   The container `OneMore f` also has an ordering, where the new element comes before any element of `f`. Write a `Foldable` instance for `OneMore f`:

   ```
   instance Foldable f => Foldable (OneMore f) where
     ...
   ```

7. (Medium) Write a `dedupShapes :: Array Shape -> Array Shape` function that removes duplicate `Shape`s from an array using the `nubEq` function.

8. (Medium) Write a `dedupShapesFast` function which is the same as `dedupShapes`, but uses the more efficient `nub` function.

Multi-Parameter Type Classes

It's not the case that a type class can only take a single type as an argument. This is the most common case, but a type class can be parameterized by *zero or more* type arguments.

Let's see an example of a type class with two type arguments.

```
module Stream where

import Data.Array as Array
import Data.Maybe (Maybe)
import Data.String.CodeUnits as String

class Stream stream element where
  uncons :: stream -> Maybe { head :: element, tail :: stream }

instance Stream (Array a) a where
  uncons = Array.uncons

instance Stream String Char where
  uncons = String.uncons
```

The `Stream` module defines a class `Stream` which identifies types that look like streams of elements, where elements can be pulled from the front of the stream using the `uncons` function.

Note that the `Stream` type class is parameterized not only by the type of the stream itself, but also by its elements. This allows us to define type class instances for the same stream type but different element types.

The module defines two type class instances: an instance for arrays, where `uncons` removes the head element of the array using pattern matching, and an instance for String, which removes the first character from a String.

We can write functions that work over arbitrary streams. For example, here is a function that accumulates a result in some `Monoid` based on the elements of a stream:

```
import Prelude
import Data.Monoid (class Monoid, mempty)

foldStream :: forall l e m. Stream l e => Monoid m => (e -> m) -> l -> m
foldStream f list =
  case uncons list of
    Nothing -> mempty
    Just cons -> f cons.head <> foldStream f cons.tail
```

Try using `foldStream` in PSCi for different types of `Stream` and different types of `Monoid`.

Functional Dependencies

Multi-parameter type classes can be very useful but can easily lead to confusing types and

even issues with type inference. As a simple example, consider writing a generic `tail` function on streams using the `Stream` class given above:

```
genericTail xs = map _.tail (uncons xs)
```

This gives a somewhat confusing error message:

```
The inferred type

  forall stream a. Stream stream a => stream -> Maybe stream

has type variables which are not mentioned in the body of the type. Consider
adding a type annotation.
```

The problem is that the `genericTail` function does not use the `element` type mentioned in the definition of the `Stream` type class, so that type is left unsolved.

Worse still, we cannot even use `genericTail` by applying it to a specific type of stream:

```
> map _.tail (uncons "testing")

The inferred type

  forall a. Stream String a => Maybe String

has type variables which are not mentioned in the body of the type. Consider
adding a type annotation.
```

Here, we might expect the compiler to choose the `streamString` instance. After all, a `String` is a stream of `Char` s, and cannot be a stream of any other type of elements.

The compiler cannot make that deduction automatically or commit to the `streamString` instance. However, we can help the compiler by adding a hint to the type class definition:

```
class Stream stream element | stream -> element where
  uncons :: stream -> Maybe { head :: element, tail :: stream }
```

Here, `stream -> element` is called a *functional dependency*. A functional dependency asserts a functional relationship between the type arguments of a multi-parameter type class. This functional dependency tells the compiler that there is a function from stream types to (unique) element types, so if the compiler knows the stream type, then it can commit to the element type.

This hint is enough for the compiler to infer the correct type for our generic tail function

above:

```
> :type genericTail
forall (stream :: Type) (element :: Type). Stream stream element => stream ->
Maybe stream

> genericTail "testing"
(Just "esting")
```

Functional dependencies can be useful when designing certain APIs using multi-parameter type classes.

Nullary Type Classes

We can even define type classes with zero-type arguments! These correspond to compile-time assertions about our functions, allowing us to track the global properties of our code in the type system.

An important example is the `Partial` class we saw earlier when discussing partial functions. Take, for example, the functions `head` and `tail` defined in `Data.Array.Partial` that allow us to get the head or tail of an array without wrapping them in a `Maybe`, so they can fail if the array is empty:

```
head :: forall a. Partial => Array a -> a

tail :: forall a. Partial => Array a -> Array a
```

Note that there is no instance defined for the `Partial` type class! Doing so would defeat its purpose: attempting to use the `head` function directly will result in a type error:

```
> head [1, 2, 3]

No type class instance was found for

  Prim.Partial
```

Instead, we can republish the `Partial` constraint for any functions making use of partial functions:

```
secondElement :: forall a. Partial => Array a -> a
secondElement xs = head (tail xs)
```

We've already seen the `unsafePartial` function, which allows us to treat a partial function as a regular function (unsafely). This function is defined in the `Partial.Unsafe` module:

```
unsafePartial :: forall a. (Partial => a) -> a
```

Note that the `Partial` constraint appears *inside the parentheses* on the left of the function arrow, but not in the outer `forall`. That is, `unsafePartial` is a function from partial values to regular values:

```
> unsafePartial head [1, 2, 3]
1

> unsafePartial secondElement [1, 2, 3]
2
```

Superclasses

Just as we can express relationships between type class instances by making an instance dependent on another instance, we can express relationships between type classes themselves using so-called *superclasses*.

We say that one type class is a superclass of another if every instance of the second class is required to be an instance of the first, and we indicate a superclass relationship in the class definition by using a backwards facing double arrow (`<=`).

We've already seen an example of superclass relationships: the `Eq` class is a superclass of `Ord`, and the `Semigroup` class is a superclass of `Monoid`. For every type class instance of the `Ord` class, there must be a corresponding `Eq` instance for the same type. This makes sense since, in many cases, when the `compare` function reports that two values are incomparable, we often want to use the `Eq` class to determine if they are equal.

In general, it makes sense to define a superclass relationship when the laws for the subclass mention the superclass members. For example, for any pair of `Ord` and `Eq` instances, it is reasonable to assume that if two values are equal under the `Eq` instance, then the `compare` function should return `EQ`. In other words, `a == b` should be true exactly when `compare a b` evaluates to `EQ`. This relationship on the level of laws justifies the superclass relationship between `Eq` and `Ord`.

Another reason to define a superclass relationship is when there is a clear "is-a" relationship between the two classes. That is, every member of the subclass *is a* member of the

superclass as well.

Exercises

1. (Medium) Define a partial function `unsafeMaximum :: Partial => Array Int -> Int` that finds the maximum of a non-empty array of integers. Test out your function in PSCi using `unsafePartial`. *Hint*: Use the `maximum` function from `Data.Foldable`.

2. (Medium) The `Action` class is a multi-parameter type class that defines an action of one type on another:

   ```
   class Monoid m <= Action m a where
     act :: m -> a -> a
   ```

   An *action* is a function that describes how monoidal values are used to determine how to modify a value of another type. There are two laws for the `Action` type class:

   - `act mempty a = a`
   - `act (m1 <> m2) a = act m1 (act m2 a)`

   Applying an empty action is a no-op. And applying two actions in sequence is the same as applying the actions combined. That is, actions respect the operations defined by the `Monoid` class.

   For example, the natural numbers form a monoid under multiplication:

   ```
   newtype Multiply = Multiply Int

   instance Semigroup Multiply where
     append (Multiply n) (Multiply m) = Multiply (n * m)

   instance Monoid Multiply where
     mempty = Multiply 1
   ```

   Write an instance that implements this action:

   ```
   instance Action Multiply Int where
     ...
   ```

Remember, your instance must satisfy the laws listed above.

3. (Difficult) There are multiple ways to implement an instance of `Action Multiply Int`. How many can you think of? PureScript does not allow multiple implementations of the same instance, so you will have to replace your original implementation. *Note*: the tests cover 4 implementations.

4. (Medium) Write an `Action` instance that repeats an input string some number of times:

   ```
   instance Action Multiply String where
     ...
   ```

   Hint: Search Pursuit for a helper-function with the signature `String -> Int -> String`. Note that `String` might appear as a more generic type (such as `Monoid`).

   Does this instance satisfy the laws listed above?

5. (Medium) Write an instance `Action m a => Action m (Array a)`, where the action on arrays is defined by acting on each array element independently.

6. (Difficult) Given the following newtype, write an instance for `Action m (Self m)`, where the monoid `m` acts on itself using `append`:

   ```
   newtype Self m = Self m
   ```

   Note: The testing framework requires `Show` and `Eq` instances for the `Self` and `Multiply` types. You may either write these instances manually, or let the compiler handle this for you with `derive newtype instance` shorthand.

7. (Difficult) Should the arguments of the multi-parameter type class `Action` be related by some functional dependency? Why or why not? *Note*: There is no test for this exercise.

A Type Class for Hashes

In the last section of this chapter, we will use the lessons from the rest of the chapter to create a library for hashing data structures.

> Note that this library is for demonstration purposes only and is not intended to provide a robust hashing mechanism.

What properties might we expect of a hash function?

- A hash function should be deterministic and map equal values to equal hash codes.
- A hash function should distribute its results approximately uniformly over some set of hash codes.

The first property looks a lot like a law for a type class, whereas the second property is more along the lines of an informal contract and certainly would not be enforceable by PureScript's type system. However, this should provide the intuition for the following type class:

```
newtype HashCode = HashCode Int

instance Eq HashCode where
  eq (HashCode a) (HashCode b) = a == b

hashCode :: Int -> HashCode
hashCode h = HashCode (h `mod` 65535)

class Eq a <= Hashable a where
  hash :: a -> HashCode
```

with the associated law that `a == b` implies `hash a == hash b`.

We'll spend the rest of this section building a library of instances and functions associated with the `Hashable` type class.

We will need a way to combine hash codes in a deterministic way:

```
combineHashes :: HashCode -> HashCode -> HashCode
combineHashes (HashCode h1) (HashCode h2) = hashCode (73 * h1 + 51 * h2)
```

The `combineHashes` function will mix two hash codes and redistribute the result over the interval 0-65535.

Let's write a function that uses the `Hashable` constraint to restrict the types of its inputs. One common task which requires a hashing function is to determine if two values hash to the same hash code. The `hashEqual` relation provides such a capability:

```
hashEqual :: forall a. Hashable a => a -> a -> Boolean
hashEqual = eq `on` hash
```

This function uses the `on` function from `Data.Function` to define hash-equality in terms of equality of hash codes, and should read like a declarative definition of hash-equality: two values are "hash-equal" if they are equal after each value passed through the `hash` function.

Let's write some `Hashable` instances for some primitive types. Let's start with an instance for integers. Since a `HashCode` is really just a wrapped integer, this is simple – we can use the `hashCode` helper function:

```
instance Hashable Int where
  hash = hashCode
```

We can also define a simple instance for `Boolean` values using pattern matching:

```
instance Hashable Boolean where
  hash false = hashCode 0
  hash true  = hashCode 1
```

With an instance for hashing integers, we can create an instance for hashing `Char` s by using the `toCharCode` function from `Data.Char` :

```
instance Hashable Char where
  hash = hash <<< toCharCode
```

To define an instance for arrays, we can `map` the `hash` function over the elements of the array (if the element type is also an instance of `Hashable`) and then perform a left fold over the resulting hashes using the `combineHashes` function:

```
instance Hashable a => Hashable (Array a) where
  hash = foldl combineHashes (hashCode 0) <<< map hash
```

Notice how we build up instances using the simpler instances we have already written. Let's use our new `Array` instance to define an instance for `String` s, by turning a `String` into an array of `Char` s:

```
instance Hashable String where
  hash = hash <<< toCharArray
```

How can we prove that these `Hashable` instances satisfy the type class law that we stated above? We need to make sure that equal values have equal hash codes. In cases like `Int` ,

`Char`, `String`, and `Boolean`, this is simple because there are no values of those types that are equal in the sense of `Eq` but not equal identically.

What about some more interesting types? To prove the type class law for the `Array` instance, we can use induction on the length of the array. The only array with a length zero is `[]`. Any two non-empty arrays are equal only if they have equal head elements and equal tails, by the definition of `Eq` on arrays. By the inductive hypothesis, the tails have equal hashes, and we know that the head elements have equal hashes if the `Hashable a` instance must satisfy the law. Therefore, the two arrays have equal hashes, and so the `Hashable (Array a)` obeys the type class law as well.

The source code for this chapter includes several other examples of `Hashable` instances, such as instances for the `Maybe` and `Tuple` type.

Exercises

1. (Easy) Use PSCi to test the hash functions for each of the defined instances. *Note*: There is no provided unit test for this exercise.

2. (Medium) Write a function `arrayHasDuplicates`, which tests if an array has any duplicate elements based on both hash and value equality. First, check for hash equality with the `hashEqual` function, then check for value equality with `==` if a duplicate pair of hashes is found. *Hint*: the `nubByEq` function in `Data.Array` should make this task much simpler.

3. (Medium) Write a `Hashable` instance for the following newtype which satisfies the type class law:

   ```
   newtype Hour = Hour Int

   instance Eq Hour where
     eq (Hour n) (Hour m) = mod n 12 == mod m 12
   ```

   The newtype `Hour` and its `Eq` instance represent the type of integers modulo 12, so that 1 and 13 are identified as equal, for example. Prove that the type class law holds for your instance.

4. (Difficult) Prove the type class laws for the `Hashable` instances for `Maybe`, `Either` and `Tuple`. *Note*: There is no test for this exercise.

Conclusion

In this chapter, we've been introduced to *type classes*, a type-oriented form of abstraction that enables powerful forms of code reuse. We've seen a collection of standard type classes from the PureScript standard libraries and defined our own library based on a type class for computing hash codes.

This chapter also introduced type class laws, a technique for proving properties about code that uses type classes for abstraction. Type class laws are part of a larger subject called *equational reasoning*, in which the properties of a programming language and its type system are used to enable logical reasoning about its programs. This is an important idea and a theme that we will return to throughout the rest of the book.

Applicative Validation

Chapter Goals

In this chapter, we will meet an important new abstraction – the *applicative functor*, described by the `Applicative` type class. Don't worry if the name sounds confusing – we will motivate the concept with a practical example – validating form data. This technique allows us to convert code which usually involves a lot of boilerplate checking into a simple, declarative description of our form.

We will also meet another type class, `Traversable`, which describes *traversable functors*, and see how this concept also arises very naturally from solutions to real-world problems.

The example code for this chapter will be a continuation of the address book example from Chapter 3. This time, we will extend our address book data types and write functions to validate values for those types. The understanding is that these functions could be used, for example, in a web user interface, to display errors to the user as part of a data entry form.

Project Setup

The source code for this chapter is defined in the files `src/Data/AddressBook.purs` and `src/Data/AddressBook/Validation.purs`.

The project has a number of dependencies, many of which we have seen before. There are two new dependencies:

- `control`, which defines functions for abstracting control flow using type classes like `Applicative`.
- `validation`, which defines a functor for *applicative validation*, the subject of this chapter.

The `Data.AddressBook` module defines data types and `Show` instances for the types in our project and the `Data.AddressBook.Validation` module contains validation rules for those types.

Generalizing Function Application

To explain the concept of an *applicative functor*, let's consider the type constructor `Maybe` that we met earlier.

The source code for this module defines a function `address` that has the following type:

```
address :: String -> String -> String -> Address
```

This function is used to construct a value of type `Address` from three strings: a street name, a city, and a state.

We can apply this function easily and see the result in PSCi:

```
> import Data.AddressBook

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

However, suppose we did not necessarily have a street, city, or state, and wanted to use the `Maybe` type to indicate a missing value in each of the three cases.

In one case, we might have a missing city. If we try to apply our function directly, we will receive an error from the type checker:

```
> import Data.Maybe
> address (Just "123 Fake St.") Nothing (Just "CA")

Could not match type

  Maybe String

with type

  String
```

Of course, this is an expected type error – `address` takes strings as arguments, not values of type `Maybe String`.

However, it is reasonable to expect that we should be able to "lift" the `address` function to work with optional values described by the `Maybe` type. In fact, we can, and the `Control.Apply` provides the function `lift3` function which does exactly what we need:

```
> import Control.Apply
> lift3 address (Just "123 Fake St.") Nothing (Just "CA")

Nothing
```

In this case, the result is `Nothing`, because one of the arguments (the city) was missing. If we provide all three arguments using the `Just` constructor, then the result will contain a value as well:

```
> lift3 address (Just "123 Fake St.") (Just "Faketown") (Just "CA")

Just ({ street: "123 Fake St.", city: "Faketown", state: "CA" })
```

The name of the function `lift3` indicates that it can be used to lift functions of 3 arguments. There are similar functions defined in `Control.Apply` for functions of other numbers of arguments.

Lifting Arbitrary Functions

So, we can lift functions with small numbers of arguments by using `lift2`, `lift3`, etc. But how can we generalize this to arbitrary functions?

It is instructive to look at the type of `lift3`:

```
> :type lift3
forall (a :: Type) (b :: Type) (c :: Type) (d :: Type) (f :: Type -> Type).
Apply f => (a -> b -> c -> d) -> f a -> f b -> f c -> f d
```

In the `Maybe` example above, the type constructor `f` is `Maybe`, so that `lift3` is specialized to the following type:

```
forall a b c d. (a -> b -> c -> d) -> Maybe a -> Maybe b -> Maybe c -> Maybe d
```

This type says that we can take any function with three arguments and lift it to give a new function whose argument and result types are wrapped with `Maybe`.

Certainly, this is not possible for every type constructor `f`, so what is it about the `Maybe` type which allowed us to do this? Well, in specializing the type above, we removed a type class constraint on `f` from the `Apply` type class. `Apply` is defined in the Prelude as follows:

```
class Functor f where
  map :: forall a b. (a -> b) -> f a -> f b

class Functor f <= Apply f where
  apply :: forall a b. f (a -> b) -> f a -> f b
```

The `Apply` type class is a subclass of `Functor`, and defines an additional function `apply`. As `<$>` was defined as an alias for `map`, the `Prelude` module defines `<*>` as an alias for `apply`. As we'll see, these two operators are often used together.

Note that this `apply` is different than the `apply` from `Data.Function` (infixed as `$`). Luckily, infix notation is almost always used for the latter, so you don't need to worry about name collisions.

The type of `apply` looks a lot like the type of `map`. The difference between `map` and `apply` is that `map` takes a function as an argument, whereas the first argument to `apply` is wrapped in the type constructor `f`. We'll see how this is used soon, but first, let's see how to implement the `Apply` type class for the `Maybe` type:

```
instance Functor Maybe where
  map f (Just a) = Just (f a)
  map f Nothing  = Nothing

instance Apply Maybe where
  apply (Just f) (Just x) = Just (f x)
  apply _        _        = Nothing
```

This type class instance says that we can apply an optional function to an optional value, and the result is defined only if both are defined.

Now we'll see how `map` and `apply` can be used together to lift functions of an arbitrary number of arguments.

For functions of one argument, we can use `map` directly.

For functions of two arguments, we have a curried function `g` with type `a -> b -> c`, say. This is equivalent to the type `a -> (b -> c)`, so we can apply `map` to `g` to get a new function of type `f a -> f (b -> c)` for any type constructor `f` with a `Functor` instance. Partially applying this function to the first lifted argument (of type `f a`), we get a new wrapped function of type `f (b -> c)`. If we also have an `Apply` instance for `f`, we can then use `apply` to apply the second lifted argument (of type `f b`) to get our final value of type `f c`.

Putting this all together, we see that if we have values `x :: f a` and `y :: f b`, then the

expression `(g <$> x) <*> y` has type `f c` (remember, this expression is equivalent to `apply (map g x) y`). The precedence rules defined in the Prelude allow us to remove the parentheses: `g <$> x <*> y`.

In general, we can use `<$>` on the first argument, and `<*>` for the remaining arguments, as illustrated here for `lift3`:

```
lift3 :: forall a b c d f
       . Apply f
      => (a -> b -> c -> d)
      -> f a
      -> f b
      -> f c
      -> f d
lift3 f x y z = f <$> x <*> y <*> z
```

It is left as an exercise for the reader to verify the types involved in this expression.

As an example, we can try lifting the address function over `Maybe`, directly using the `<$>` and `<*>` functions:

```
> address <$> Just "123 Fake St." <*> Just "Faketown" <*> Just "CA"
Just ({ street: "123 Fake St.", city: "Faketown", state: "CA" })

> address <$> Just "123 Fake St." <*> Nothing <*> Just "CA"
Nothing
```

Try lifting some other functions of various numbers of arguments over `Maybe` in this way.

Alternatively, *applicative do notation* can be used for the same purpose in a way that looks similar to the familiar *do notation*. Here is `lift3` using *applicative do notation*. Note `ado` is used instead of `do`, and `in` is used on the final line to denote the yielded value:

```
lift3 :: forall a b c d f
       . Apply f
      => (a -> b -> c -> d)
      -> f a
      -> f b
      -> f c
      -> f d
lift3 f x y z = ado
  a <- x
  b <- y
  c <- z
  in f a b c
```

The Applicative Type Class

There is a related type class called `Applicative`, defined as follows:

```
class Apply f <= Applicative f where
  pure :: forall a. a -> f a
```

`Applicative` is a subclass of `Apply` and defines the `pure` function. `pure` takes a value and returns a value whose type has been wrapped with the type constructor `f`.

Here is the `Applicative` instance for `Maybe`:

```
instance Applicative Maybe where
  pure x = Just x
```

If we think of applicative functors as functors that allow lifting of functions, then `pure` can be thought of as lifting functions of zero arguments.

Intuition for Applicative

Functions in PureScript are pure and do not support side-effects. Applicative functors allow us to work in larger "programming languages" which support some sort of side-effect encoded by the functor `f`.

As an example, the functor `Maybe` represents the side effect of possibly-missing values. Some other examples include `Either err`, which represents the side effect of possible errors of type `err`, and the arrow functor `r ->`, which represents the side-effect of reading

from a global configuration. For now, we'll only consider the `Maybe` functor.

If the functor `f` represents this larger programming language with effects, then the `Apply` and `Applicative` instances allow us to lift values and function applications from our smaller programming language (PureScript) into the new language.

`pure` lifts pure (side-effect free) values into the larger language; for functions, we can use `map` and `apply` as described above.

This raises a question: if we can use `Applicative` to embed PureScript functions and values into this new language, then how is the new language any larger? The answer depends on the functor `f`. If we can find expressions of type `f a` which cannot be expressed as `pure x` for some `x`, then that expression represents a term which only exists in the larger language.

When `f` is `Maybe`, an example is the expression `Nothing`: we cannot write `Nothing` as `pure x` for any `x`. Therefore, we can think of PureScript as having been enlarged to include the new term `Nothing`, which represents a missing value.

More Effects

Let's see some more examples of lifting functions over different `Applicative` functors.

Here is a simple example function defined in PSCi, which joins three names to form a full name:

```
> import Prelude

> fullName first middle last = last <> ", " <> first <> " " <> middle

> fullName "Phillip" "A" "Freeman"
Freeman, Phillip A
```

Suppose that this function forms the implementation of a (very simple!) web service with the three arguments provided as query parameters. We want to ensure that the user provided each of the three parameters, so we might use the Maybe type to indicate the presence or absence of a parameter. We can lift `fullName` over `Maybe` to create an implementation of the web service which checks for missing parameters:

```
> import Data.Maybe

> fullName <$> Just "Phillip" <*> Just "A" <*> Just "Freeman"
Just ("Freeman, Phillip A")

> fullName <$> Just "Phillip" <*> Nothing <*> Just "Freeman"
Nothing
```

Or with *applicative do*:

```
> import Data.Maybe

> :paste…
… ado
…   f <- Just "Phillip"
…   m <- Just "A"
…   l <- Just "Freeman"
…   in fullName f m l
… ^D
(Just "Freeman, Phillip A")

… ado
…   f <- Just "Phillip"
…   m <- Nothing
…   l <- Just "Freeman"
…   in fullName f m l
… ^D
Nothing
```

Note that the lifted function returns `Nothing` if any of the arguments was `Nothing`.

This is good because now we can send an error response back from our web service if the parameters are invalid. However, it would be better if we could indicate which field was incorrect in the response.

Instead of lifting over `Maybe`, we can lift over `Either String`, which allows us to return an error message. First, let's write an operator to convert optional inputs into computations which can signal an error using `Either String`:

```
> import Data.Either
> :paste
… withError Nothing  err = Left err
… withError (Just a) _   = Right a
… ^D
```

Note: In the `Either err` applicative functor, the `Left` constructor indicates an error, and the `Right` constructor indicates success.

Now we can lift over `Either String`, providing an appropriate error message for each parameter:

```
> :paste
… fullNameEither first middle last =
…   fullName <$> (first  `withError` "First name was missing")
…            <*> (middle `withError` "Middle name was missing")
…            <*> (last   `withError` "Last name was missing")
… ^D
```

Or with *applicative do*:

```
> :paste
… fullNameEither first middle last = ado
…  f <- first  `withError` "First name was missing"
…  m <- middle `withError` "Middle name was missing"
…  l <- last   `withError` "Last name was missing"
…  in fullName f m l
… ^D

> :type fullNameEither
Maybe String -> Maybe String -> Maybe String -> Either String String
```

Now our function takes three optional arguments using `Maybe`, and returns either a `String` error message or a `String` result.

We can try out the function with different inputs:

```
> fullNameEither (Just "Phillip") (Just "A") (Just "Freeman")
(Right "Freeman, Phillip A")

> fullNameEither (Just "Phillip") Nothing (Just "Freeman")
(Left "Middle name was missing")

> fullNameEither (Just "Phillip") (Just "A") Nothing
(Left "Last name was missing")
```

In this case, we see the error message corresponding to the first missing field or a successful result if every field was provided. However, if we are missing multiple inputs, we still only see the first error:

```
> fullNameEither Nothing Nothing Nothing
(Left "First name was missing")
```

This might be good enough, but if we want to see a list of *all* missing fields in the error, then we need something more powerful than `Either String`. We will see a solution later in this

chapter.

Combining Effects

As an example of working with applicative functors abstractly, this section will show how to write a function that generically combines side-effects encoded by an applicative functor `f`.

What does this mean? Well, suppose we have a list of wrapped arguments of type `f a` for some `a`. That is, suppose we have a list of type `List (f a)`. Intuitively, this represents a list of computations with side-effects tracked by `f`, each with return type `a`. If we could run all of these computations in order, we would obtain a list of results of type `List a`. However, we would still have side-effects tracked by `f`. That is, we expect to be able to turn something of type `List (f a)` into something of type `f (List a)` by "combining" the effects inside the original list.

For any fixed list size `n`, there is a function of `n` arguments that builds a list of size `n` out of those arguments. For example, if `n` is `3`, the function is `\x y z -> x : y : z : Nil`. This function has type `a -> a -> a -> List a`. We can use the `Applicative` instance for `List` to lift this function over `f`, to get a function of type `f a -> f a -> f a -> f (List a)`. But, since we can do this for any `n`, it makes sense that we should be able to perform the same lifting for any *list* of arguments.

That means that we should be able to write a function

```
combineList :: forall f a. Applicative f => List (f a) -> f (List a)
```

This function will take a list of arguments, which possibly have side-effects, and return a single wrapped list, applying the side-effects of each.

To write this function, we'll consider the length of the list of arguments. If the list is empty, then we do not need to perform any effects, and we can use `pure` to simply return an empty list:

```
combineList Nil = pure Nil
```

In fact, this is the only thing we can do!

If the list is non-empty, then we have a head element, which is a wrapped argument of type `f a`, and a tail of type `List (f a)`. We can recursively combine the effects in the tail, giving a result of type `f (List a)`. We can then use `<$>` and `<*>` to lift the `Cons` constructor

over the head and new tail:

```
combineList (Cons x xs) = Cons <$> x <*> combineList xs
```

Again, this was the only sensible implementation, based on the types we were given.

We can test this function in PSCi, using the `Maybe` type constructor as an example:

```
> import Data.List
> import Data.Maybe

> combineList (fromFoldable [Just 1, Just 2, Just 3])
(Just (Cons 1 (Cons 2 (Cons 3 Nil))))

> combineList (fromFoldable [Just 1, Nothing, Just 2])
Nothing
```

When specialized to `Maybe`, our function returns a `Just` only if every list element is `Just`; otherwise, it returns `Nothing`. This is consistent with our intuition of working in a larger language supporting optional values – a list of computations that produce optional results only has a result itself if every computation contained a result.

But the `combineList` function works for any `Applicative`! We can use it to combine computations that possibly signal an error using `Either err`, or which read from a global configuration using `r ->`.

We will see the `combineList` function again later when we consider `Traversable` functors.

Exercises

1. (Medium) Write versions of the numeric operators `+`, `-`, `*`, and `/` which work with optional arguments (i.e., arguments wrapped in `Maybe`) and return a value wrapped in `Maybe`. Name these functions `addMaybe`, `subMaybe`, `mulMaybe`, and `divMaybe`. *Hint*: Use `lift2`.
2. (Medium) Extend the above exercise to work with all `Apply` types (not just `Maybe`). Name these new functions `addApply`, `subApply`, `mulApply`, and `divApply`.
3. (Difficult) Write a function `combineMaybe` which has type `forall a f. Applicative f => Maybe (f a) -> f (Maybe a)`. This function takes an optional computation with side-effects and returns a side-effecting computation with an optional result.

Applicative Validation

The source code for this chapter defines several data types which might be used in an address book application. The details are omitted here, but the key functions exported by the `Data.AddressBook` module have the following types:

```
address :: String -> String -> String -> Address

phoneNumber :: PhoneType -> String -> PhoneNumber

person :: String -> String -> Address -> Array PhoneNumber -> Person
```

Where `PhoneType` is defined as an algebraic data type:

```
data PhoneType
  = HomePhone
  | WorkPhone
  | CellPhone
  | OtherPhone
```

These functions can construct a `Person` representing an address book entry. For example, the following value is defined in `Data.AddressBook`:

```
examplePerson :: Person
examplePerson =
  person "John" "Smith"
    (address "123 Fake St." "FakeTown" "CA")
    [ phoneNumber HomePhone "555-555-5555"
    , phoneNumber CellPhone "555-555-0000"
    ]
```

Test this value in PSCi (this result has been formatted):

```
> import Data.AddressBook

> examplePerson
{ firstName: "John"
, lastName: "Smith"
, homeAddress:
    { street: "123 Fake St."
    , city: "FakeTown"
    , state: "CA"
    }
, phones:
    [ { type: HomePhone
      , number: "555-555-5555"
      }
    , { type: CellPhone
      , number: "555-555-0000"
      }
    ]
}
```

We saw in a previous section how we could use the `Either String` functor to validate a data structure of type `Person`. For example, provided functions to validate the two names in the structure, we might validate the entire data structure as follows:

```
nonEmpty1 :: String -> Either String String
nonEmpty1 ""     = Left "Field cannot be empty"
nonEmpty1 value  = Right value

validatePerson1 :: Person -> Either String Person
validatePerson1 p =
  person <$> nonEmpty1 p.firstName
         <*> nonEmpty1 p.lastName
         <*> pure p.homeAddress
         <*> pure p.phones
```

Or with *applicative do*:

```
validatePerson1Ado :: Person -> Either String Person
validatePerson1Ado p = ado
  f <- nonEmpty1 p.firstName
  l <- nonEmpty1 p.lastName
  in person f l p.homeAddress p.phones
```

In the first two lines, we use the `nonEmpty1` function to validate a non-empty string. `nonEmpty1` returns an error indicated with the `Left` constructor if its input is empty. Otherwise, it returns the value wrapped with the `Right` constructor.

The final lines do not perform any validation but simply provide the `address` and `phones`

fields to the `person` function as the remaining arguments.

This function can be seen to work in PSCi, but it has a limitation that we have seen before:

```
> validatePerson $ person "" "" (address "" "" "") []
(Left "Field cannot be empty")
```

The `Either String` applicative functor only provides the first error encountered. Given the input here, we would prefer to see two errors – one for the missing first name and a second for the missing last name.

There is another applicative functor that the `validation` library provides. This functor is called `V`, and it can return errors in any *semigroup*. For example, we can use `V (Array String)` to return an array of `String`s as errors, concatenating new errors onto the end of the array.

The `Data.AddressBook.Validation` module uses the `V (Array String)` applicative functor to validate the data structures in the `Data.AddressBook` module.

Here is an example of a validator taken from the `Data.AddressBook.Validation` module:

```
type Errors
  = Array String

nonEmpty :: String -> String -> V Errors String
nonEmpty field ""     = invalid [ "Field '" <> field <> "' cannot be empty" ]
nonEmpty _     value  = pure value

lengthIs :: String -> Int -> String -> V Errors String
lengthIs field len value | length value /= len =
  invalid [ "Field '" <> field <> "' must have length " <> show len ]
lengthIs _     _   value = pure value

validateAddress :: Address -> V Errors Address
validateAddress a =
  address <$> nonEmpty "Street"  a.street
          <*> nonEmpty "City"    a.city
          <*> lengthIs "State" 2 a.state
```

Or with *applicative do*:

```
validateAddressAdo :: Address -> V Errors Address
validateAddressAdo a = ado
  street <- nonEmpty "Street"  a.street
  city   <- nonEmpty "City"    a.city
  state  <- lengthIs "State" 2 a.state
  in address street city state
```

`validateAddress` validates an `Address` structure. It checks that the `street` and `city` fields are non-empty and that the string in the `state` field has length 2.

Notice how the `nonEmpty` and `lengthIs` validator functions both use the `invalid` function provided by the `Data.Validation` module to indicate an error. Since we are working in the `Array String` semigroup, `invalid` takes an array of strings as its argument.

We can try this function in PSCi:

```
> import Data.AddressBook
> import Data.AddressBook.Validation

> validateAddress $ address "" "" ""
(invalid [ "Field 'Street' cannot be empty"
         , "Field 'City' cannot be empty"
         , "Field 'State' must have length 2"
         ])

> validateAddress $ address "" "" "CA"
(invalid [ "Field 'Street' cannot be empty"
         , "Field 'City' cannot be empty"
         ])
```

This time, we receive an array of all validation errors.

Regular Expression Validators

The `validatePhoneNumber` function uses a regular expression to validate the form of its argument. The key is a `matches` validation function, which uses a `Regex` from the `Data.String.Regex` module to validate its input:

```
matches :: String -> Regex -> String -> V Errors String
matches _     regex value | test regex value
                          = pure value
matches field _     _     = invalid [ "Field '" <> field <> "' did not match
the required format" ]
```

Again, notice how `pure` is used to indicate successful validation, and `invalid` is used to signal an array of errors.

`validatePhoneNumber` is built from the `matches` function in the same way as before:

```
validatePhoneNumber :: PhoneNumber -> V Errors PhoneNumber
validatePhoneNumber pn =
  phoneNumber <$> pure pn."type"
              <*> matches "Number" phoneNumberRegex pn.number
```

Or with *applicative do*:

```
validatePhoneNumberAdo :: PhoneNumber -> V Errors PhoneNumber
validatePhoneNumberAdo pn = ado
  tpe    <- pure pn."type"
  number <- matches "Number" phoneNumberRegex pn.number
  in phoneNumber tpe number
```

Again, try running this validator against some valid and invalid inputs in PSCi:

```
> validatePhoneNumber $ phoneNumber HomePhone "555-555-5555"
pure ({ type: HomePhone, number: "555-555-5555" })

> validatePhoneNumber $ phoneNumber HomePhone "555.555.5555"
invalid (["Field 'Number' did not match the required format"])
```

Exercises

1. (Easy) Write a regular expression `stateRegex :: Regex` to check that a string only contains two alphabetic characters. *Hint*: see the source code for `phoneNumberRegex`.
2. (Medium) Write a regular expression `nonEmptyRegex :: Regex` to check that a string is not entirely whitespace. *Hint*: If you need help developing this regex expression, check out RegExr, which has a great cheatsheet and interactive test environment.
3. (Medium) Write a function `validateAddressImproved` that is similar to `validateAddress`, but uses the above `stateRegex` to validate the `state` field and `nonEmptyRegex` to validate the `street` and `city` fields. *Hint*: see the source for `validatePhoneNumber` for an example of how to use `matches`.

Traversable Functors

The remaining validator is `validatePerson`, which combines the validators we have seen so far to validate an entire `Person` structure, including the following new `validatePhoneNumbers` function:

```
validatePhoneNumbers :: String -> Array PhoneNumber -> V Errors (Array
PhoneNumber)
validatePhoneNumbers field []      =
  invalid [ "Field '" <> field <> "' must contain at least one value" ]
validatePhoneNumbers _     phones  =
  traverse validatePhoneNumber phones

validatePerson :: Person -> V Errors Person
validatePerson p =
  person <$> nonEmpty "First Name" p.firstName
         <*> nonEmpty "Last Name" p.lastName
         <*> validateAddress p.homeAddress
         <*> validatePhoneNumbers "Phone Numbers" p.phones
```

or with *applicative do*

```
validatePersonAdo :: Person -> V Errors Person
validatePersonAdo p = ado
  firstName <- nonEmpty "First Name" p.firstName
  lastName  <- nonEmpty "Last Name" p.lastName
  address   <- validateAddress p.homeAddress
  numbers   <- validatePhoneNumbers "Phone Numbers" p.phones
  in person firstName lastName address numbers
```

`validatePhoneNumbers` uses a new function we haven't seen before – `traverse`.

`traverse` is defined in the `Data.Traversable` module, in the `Traversable` type class:

```
class (Functor t, Foldable t) <= Traversable t where
  traverse :: forall a b m. Applicative m => (a -> m b) -> t a -> m (t b)
  sequence :: forall a m. Applicative m => t (m a) -> m (t a)
```

`Traversable` defines the class of *traversable functors*. The types of its functions might look a little intimidating, but `validatePerson` provides a good motivating example.

Every traversable functor is both a `Functor` and `Foldable` (recall that a *foldable functor* was a type constructor that supported a fold operation, reducing a structure to a single value). In addition, a traversable functor can combine a collection of side-effects that depend on its structure.

This may sound complicated, but let's simplify things by specializing to the case of arrays. The array type constructor is traversable, which means that there is a function:

```
traverse :: forall a b m. Applicative m => (a -> m b) -> Array a -> m (Array b)
```

Intuitively, given any applicative functor `m`, and a function which takes a value of type `a` and returns a value of type `b` (with side-effects tracked by `m`), we can apply the function to each element of an array of type `Array a` to obtain a result of type `Array b` (with side-effects tracked by `m`).

Still not clear? Let's specialize further to the case where `m` is the `V Errors` applicative functor above. Now, we have a function of type

```
traverse :: forall a b. (a -> V Errors b) -> Array a -> V Errors (Array b)
```

This type signature says that if we have a validation function `m` for a type `a`, then `traverse m` is a validation function for arrays of type `Array a`. But that's exactly what we need to be able to validate the `phones` field of the `Person` data structure! We pass `validatePhoneNumber` to `traverse` to create a validation function that validates each element successively.

In general, `traverse` walks over the elements of a data structure, performing computations with side-effects and accumulating a result.

The type signature for `Traversable`'s other function `sequence` might look more familiar:

```
sequence :: forall a m. Applicative m => t (m a) -> m (t a)
```

In fact, the `combineList` function that we wrote earlier is just a special case of the `sequence` function from the `Traversable` type class. Setting `t` to be the type constructor `List`, we recover the type of the `combineList` function:

```
combineList :: forall f a. Applicative f => List (f a) -> f (List a)
```

Traversable functors capture the idea of traversing a data structure, collecting a set of effectful computations, and combining their effects. In fact, `sequence` and `traverse` are equally important to the definition of `Traversable` – each can be implemented in terms of the other. This is left as an exercise for the interested reader.

The `Traversable` instance for lists given in the `Data.List` module is:

```
instance Traversable List where
-- traverse :: forall a b m. Applicative m => (a -> m b) -> List a -> m (List
b)
traverse _ Nil         = pure Nil
traverse f (Cons x xs) = Cons <$> f x <*> traverse f xs
```

(The actual definition was later modified to improve stack safety. You can read more about that change here.)

In the case of an empty list, we can return an empty list using `pure`. If the list is non-empty, we can use the function `f` to create a computation of type `f b` from the head element. We can also call `traverse` recursively on the tail. Finally, we can lift the `Cons` constructor over the applicative functor `m` to combine the two results.

But there are more examples of traversable functors than just arrays and lists. The `Maybe` type constructor we saw earlier also has an instance for `Traversable`. We can try it in PSCi:

```
> import Data.Maybe
> import Data.Traversable
> import Data.AddressBook.Validation

> traverse (nonEmpty "Example") Nothing
pure (Nothing)

> traverse (nonEmpty "Example") (Just "")
invalid (["Field 'Example' cannot be empty"])

> traverse (nonEmpty "Example") (Just "Testing")
pure ((Just "Testing"))
```

These examples show that traversing the `Nothing` value returns `Nothing` with no validation, and traversing `Just x` uses the validation function to validate `x`. That is, `traverse` takes a validation function for type `a` and returns a validation function for `Maybe a`, i.e., a validation function for optional values of type `a`.

Other traversable functors include `Array`, `Tuple a`, and `Either a` for any type `a`. Generally, most "container" data type constructors have `Traversable` instances. As an example, the exercises will include writing a `Traversable` instance for a type of binary trees.

Exercises

1. (Easy) Write `Eq` and `Show` instances for the following binary tree data structure:

```
data Tree a = Leaf | Branch (Tree a) a (Tree a)
```

Recall from the previous chapter that you may either write these instances manually or let the compiler derive them.

There are many "correct" formatting options for `Show` output. The test for this exercise expects the following whitespace style. This matches the default formatting of the generic show, so you only need to note this if you're planning on writing this instance manually.

```
(Branch (Branch Leaf 8 Leaf) 42 Leaf)
```

2. (Medium) Write a `Traversable` instance for `Tree a`, which combines side-effects left-to-right. *Hint*: There are some additional instance dependencies that need to be defined for `Traversable`.

3. (Medium) Write a function `traversePreOrder :: forall a m b. Applicative m => (a -> m b) -> Tree a -> m (Tree b)` that performs a pre-order traversal of the tree. This means the order of effect execution is root-left-right, instead of left-root-right as was done for the previous in-order traverse exercise. *Hint*: No additional instances need to be defined, and you don't need to call any of the functions defined earlier. Applicative do notation (`ado`) is the easiest way to write this function.

4. (Medium) Write a function `traversePostOrder` that performs a post-order traversal of the tree where effects are executed left-right-root.

5. (Medium) Create a new version of the `Person` type where the `homeAddress` field is optional (using `Maybe`). Then write a new version of `validatePerson` (renamed as `validatePersonOptionalAddress`) to validate this new `Person`. *Hint*: Use `traverse` to validate a field of type `Maybe a`.

6. (Difficult) Write a function `sequenceUsingTraverse` which behaves like `sequence`, but is written in terms of `traverse`.

7. (Difficult) Write a function `traverseUsingSequence` which behaves like `traverse`, but is written in terms of `sequence`.

Applicative Functors for Parallelism

In the discussion above, I chose the word "combine" to describe how applicative functors "combine side-effects". However, in all the examples given, it would be equally valid to say that applicative functors allow us to "sequence" effects. This would be consistent with the intuition that traversable functors provide a `sequence` function to combine effects in sequence based on a data structure.

However, in general, applicative functors are more general than this. The applicative functor laws do not impose any ordering on the side-effects that their computations perform. It would be valid for an applicative functor to perform its side-effects in parallel.

For example, the `V` validation functor returned an *array* of errors, but it would work just as well if we picked the `Set` semigroup, in which case it would not matter what order we ran the various validators. We could even run them in parallel over the data structure!

As a second example, the `parallel` package provides a type class `Parallel` which supports *parallel computations*. `Parallel` provides a function `parallel` that uses some `Applicative` functor to compute the result of its input computation *in parallel*:

```
f <$> parallel computation1
  <*> parallel computation2
```

This computation would start computing values asynchronously using `computation1` and `computation2`. When both results have been computed, they would be combined into a single result using the function `f`.

We will see this idea in more detail when we apply applicative functors to the problem of *callback hell* later in the book.

Applicative functors are a natural way to capture side-effects that can be combined in parallel.

Conclusion

In this chapter, we covered a lot of new ideas:

- We introduced the concept of an *applicative functor* which generalizes the idea of function application to type constructors that captures some notion of side-effect.
- We saw how applicative functors solved the problem of validating data structures and how by switching the applicative functor, we could change from reporting a single error to reporting all errors across a data structure.
- We met the `Traversable` type class, which encapsulates the idea of a *traversable*

functor, or a container whose elements can be used to combine values with side-effects.

Applicative functors are an interesting abstraction that provides neat solutions to a number of problems. We will see them a few more times throughout the book. In this case, the validation applicative functor provided a way to write validators in a declarative style, allowing us to define *what* our validators should validate and not *how* they should perform that validation. In general, we will see that applicative functors are a useful tool for the design of *domain specific languages*.

In the next chapter, we will see a related idea, the class of *monads*, and extend our address book example to run in the browser!

The Effect Monad

Chapter Goals

In the last chapter, we introduced applicative functors, an abstraction we used to deal with *side-effects*: optional values, error messages, and validation. This chapter will introduce another abstraction for dealing with side-effects more expressively: *monads*.

The goal of this chapter is to explain why monads are a useful abstraction and their connection with *do notation*.

Project Setup

The project adds the following dependencies:

- `effect` – defines the `Effect` monad, the subject of the second half of the chapter. This dependency is often listed in every starter project (it's been a dependency of every chapter so far), so you'll rarely have to install it explicitly.
- `react-basic-hooks` – a web framework we will use for our Address Book app.

Monads and Do Notation

Do notation was first introduced when we covered *array comprehensions*. Array comprehensions provide syntactic sugar for the `concatMap` function from the `Data.Array` module.

Consider the following example. Suppose we throw two dice and want to count the number of ways in which we can score a total of `n`. We could do this using the following non-deterministic algorithm:

- *Choose* the value `x` of the first throw.
- *Choose* the value `y` of the second throw.
- If the sum of `x` and `y` is `n`, return the pair `[x, y]`, else fail.

Array comprehensions allow us to write this non-deterministic algorithm naturally:

```
import Prelude

import Control.Plus (empty)
import Data.Array ((..))

countThrows :: Int -> Array (Array Int)
countThrows n = do
  x <- 1 .. 6
  y <- 1 .. 6
  if x + y == n
    then pure [ x, y ]
    else empty
```

We can see that this function works in PSCi:

```
> import Test.Examples

> countThrows 10
[[4,6],[5,5],[6,4]]

> countThrows 12
[[6,6]]
```

In the last chapter, we formed an intuition for the `Maybe` applicative functor, embedding PureScript functions into a larger programming language supporting *optional values*. In the same way, we can form an intuition for the *array monad*, embedding PureScript functions into a larger programming language supporting *non-deterministic choice*.

Generally, a *monad* for some type constructor `m` provides a way to use do notation with values of type `m a`. Note that in the array comprehension above, every line contains a computation of type `Array a` for some type `a`. In general, every line of a do notation block will contain a computation of type `m a` for some type `a` and our monad `m`. The monad `m` must be the same on every line (i.e., we fix the side-effect), but the types `a` can differ (i.e., individual computations can have different result types).

Here is another example of do notation, this time applied to the type constructor `Maybe`. Suppose we have some type `XML` representing XML nodes, and a function

```
child :: XML -> String -> Maybe XML
```

Which looks for a child element of a node and returns `Nothing` if no such element exists.

In this case, we can look for a deeply-nested element using do notation. Suppose we wanted to read a user's city from a user profile that had been encoded as an XML document:

```
userCity :: XML -> Maybe XML
userCity root = do
  prof <- child root "profile"
  addr <- child prof "address"
  city <- child addr "city"
  pure city
```

The `userCity` function looks for a child element `profile`, an element `address` inside the `profile` element, and finally, an element `city` inside the `address` element. If any of these elements are missing, the return value will be `Nothing`. Otherwise, the return value is constructed using `Just` from the `city` node.

Remember, the `pure` function in the last line is defined for every `Applicative` functor. Since `pure` is defined as `Just` for the `Maybe` applicative functor, it would be equally valid to change the last line to `Just city`.

The Monad Type Class

The `Monad` type class is defined as follows:

```
class Apply m <= Bind m where
  bind :: forall a b. m a -> (a -> m b) -> m b

class (Applicative m, Bind m) <= Monad m
```

The key function here is `bind`, defined in the `Bind` type class. Just like for the `<$>` and `<*>` operators in the `Functor` and `Apply` type classes, the Prelude defines an infix alias `>>=` for the `bind` function.

The `Monad` type class extends `Bind` with the operations of the `Applicative` type class we've already seen.

It will be useful to see some examples of the `Bind` type class. A sensible definition for `Bind` on arrays can be given as follows:

```
instance Bind Array where
  bind xs f = concatMap f xs
```

This explains the connection between array comprehensions and the `concatMap` function that has been alluded to before.

Here is an implementation of `Bind` for the `Maybe` type constructor:

```
instance Bind Maybe where
  bind Nothing  _ = Nothing
  bind (Just a) f = f a
```

This definition confirms the intuition that missing values are propagated through a do notation block.

Let's see how the `Bind` type class is related to do notation. Consider a simple do notation block that starts by binding a value from the result of some computation:

```
do value <- someComputation
   whatToDoNext
```

Every time the PureScript compiler sees this pattern, it replaces the code with this:

```
bind someComputation \value -> whatToDoNext
```

or, written infix:

```
someComputation >>= \value -> whatToDoNext
```

The computation `whatToDoNext` is allowed to depend on `value`.

If there are multiple binds involved, this rule is applied multiple times, starting from the top. For example, the `userCity` example that we saw earlier gets desugared as follows:

```
userCity :: XML -> Maybe XML
userCity root =
  child root "profile" >>= \prof ->
    child prof "address" >>= \addr ->
      child addr "city" >>= \city ->
        pure city
```

Notably, code expressed using do notation is often much clearer than the equivalent code using the `>>=` operator. However, writing binds explicitly using `>>=` can often lead to opportunities to write code in *point-free* form – but the usual warnings about readability apply.

Monad Laws

The `Monad` type class comes equipped with three laws, called the *monad laws*. These tell us what we can expect from sensible implementations of the `Monad` type class.

It is simplest to explain these laws using do notation.

Identity Laws

The *right-identity* law is the simplest of the three laws. It tells us that we can eliminate a call to `pure` if it is the last expression in a do notation block:

```
do
  x <- expr
  pure x
```

The right-identity law says that this is equivalent to just `expr`.

The *left-identity* law states that we can eliminate a call to `pure` if it is the first expression in a do notation block:

```
do
  x <- pure y
  next
```

This code is equivalent to `next`, after the name `x` has been replaced with the expression `y`.

The last law is the *associativity law*. It tells us how to deal with nested do notation blocks. It states that the following piece of code:

```
c1 = do
  y <- do
    x <- m1
    m2
  m3
```

is equivalent to this code:

```
c2 = do
  x <- m1
  y <- m2
  m3
```

Each of these computations involves three monadic expressions `m1`, `m2`, and `m3`. In each

case, the result of `m1` is eventually bound to the name `x`, and the result of `m2` is bound to the name `y`.

In `c1`, the two expressions `m1` and `m2` are grouped into their own do notation block.

In `c2`, all three expressions `m1`, `m2`, and `m3` appear in the same do notation block.

The associativity law tells us that it is safe to simplify nested do notation blocks in this way.

Note that by the definition of how do notation gets desugared into calls to `bind`, both of `c1` and `c2` are also equivalent to this code:

```
c3 = do
  x <- m1
  do
    y <- m2
    m3
```

Folding With Monads

As an example of working with monads abstractly, this section will present a function that works with any type constructor in the `Monad` type class. This should solidify the intuition that monadic code corresponds to programming "in a larger language" with side-effects, and also illustrate the generality which programming with monads brings.

The function we will write is called `foldM`. It generalizes the `foldl` function we met earlier to a monadic context. Here is its type signature:

```
foldM :: forall m a b. Monad m => (a -> b -> m a) -> a -> List b -> m a
foldl :: forall   a b.            (a -> b ->   a) -> a -> List b ->   a
```

Notice that this is the same as the type of `foldl`, except for the appearance of the monad `m`.

Intuitively, `foldM` performs a fold over a list in some context supporting some set of side-effects.

For example, if we picked `m` to be `Maybe`, then our fold would be allowed to fail by returning `Nothing` at any stage – every step returns an optional result, and the result of the fold is therefore also optional.

If we picked `m` to be the `Array` type constructor, then every step of the fold would be

allowed to return zero or more results, and the fold would proceed to the next step independently for each result. In the end, the set of results would consist of all folds over all possible paths. This corresponds to a traversal of a graph!

To write `foldM`, we can simply break the input list into cases.

If the list is empty, then to produce the result of type `a`, we only have one option: we have to return the second argument:

```
foldM _ a Nil = pure a
```

Note that we have to use `pure` to lift `a` into the monad `m`.

What if the list is non-empty? In that case, we have a value of type `a`, a value of type `b`, and a function of type `a -> b -> m a`. If we apply the function, we obtain a monadic result of type `m a`. We can bind the result of this computation with a backwards arrow `<-`.

It only remains to recurse on the tail of the list. The implementation is simple:

```
foldM f a (b : bs) = do
  a' <- f a b
  foldM f a' bs
```

Note that this implementation is almost identical to that of `foldl` on lists, except for do notation.

We can define and test this function in PSCi. Here is an example – suppose we defined a "safe division" function on integers, which tested for division by zero and used the `Maybe` type constructor to indicate failure:

```
safeDivide :: Int -> Int -> Maybe Int
safeDivide _ 0 = Nothing
safeDivide a b = Just (a / b)
```

Then we can use `foldM` to express iterated safe division:

```
> import Test.Examples
> import Data.List (fromFoldable)

> foldM safeDivide 100 (fromFoldable [5, 2, 2])
(Just 5)

> foldM safeDivide 100 (fromFoldable [2, 0, 4])
Nothing
```

The `foldM safeDivide` function returns `Nothing` if a division by zero was attempted at any point. Otherwise, it returns the result of repeatedly dividing the accumulator, wrapped in the `Just` constructor.

Monads and Applicatives

Every instance of the `Monad` type class is also an instance of the `Apply` type class, by virtue of the superclass relationship between the two classes.

However, there is also an implementation of the `Apply` type class which comes "for free" for any instance of `Monad`, given by the `ap` function:

```
ap :: forall m a b. Monad m => m (a -> b) -> m a -> m b
ap mf ma = do
  f <- mf
  a <- ma
  pure (f a)
```

If `m` is a law-abiding member of the `Monad` type class, then there is a valid `Apply` instance for `m` given by `ap`.

The interested reader can check that `ap` agrees with `apply` for the monads we have already encountered: `Array`, `Maybe`, and `Either e`.

If every monad is also an applicative functor, then we should be able to apply our intuition for applicative functors to every monad. In particular, we can reasonably expect a monad to correspond, in some sense, to programming "in a larger language" augmented with some set of additional side-effects. We should be able to lift functions of arbitrary arities, using `map` and `apply`, into this new language.

But monads allow us to do more than we could do with just applicative functors, and the key difference is highlighted by the syntax of do notation. Consider the `userCity` example again, in which we looked for a user's city in an XML document that encoded their user profile:

```
userCity :: XML -> Maybe XML
userCity root = do
  prof <- child root "profile"
  addr <- child prof "address"
  city <- child addr "city"
  pure city
```

Do notation allows the second computation to depend on the result `prof` of the first, and the third computation to depend on the result `addr` of the second, and so on. This dependence on previous values is not possible using only the interface of the `Applicative` type class.

Try writing `userCity` using only `pure` and `apply` : you will see that it is impossible. Applicative functors only allow us to lift function arguments which are independent of each other, but monads allow us to write computations which involve more interesting data dependencies.

In the last chapter, we saw that the `Applicative` type class can be used to express parallelism. This was precisely because the function arguments being lifted were independent of one another. Since the `Monad` type class allows computations to depend on the results of previous computations, the same does not apply – a monad has to combine its side-effects in sequence.

Exercises

1. (Easy) Write a function `third` that returns the third element of an array with three or more elements. Your function should return an appropriate `Maybe` type. *Hint:* Look up the types of the `head` and `tail` functions from the `Data.Array` module in the `arrays` package. Use do notation with the `Maybe` monad to combine these functions.

2. (Medium) Write a function `possibleSums` which uses `foldM` to determine all possible totals that could be made using a set of coins. The coins will be specified as an array which contains the value of each coin. Your function should have the following result:

   ```
   > possibleSums []
   [0]

   > possibleSums [1, 2, 10]
   [0,1,2,3,10,11,12,13]
   ```

   Hint: This function can be written as a one-liner using `foldM` . You might want to use the `nub` and `sort` functions to remove duplicates and sort the result.

3. (Medium) Confirm that the `ap` function and the `apply` operator agree for the `Maybe` monad. *Note:* There are no tests for this exercise.

4. (Medium) Verify that the monad laws hold for the `Monad` instance for the `Maybe` type,

as defined in the `maybe` package. *Note:* There are no tests for this exercise.

5. (Medium) Write a function `filterM` which generalizes the `filter` function on lists. Your function should have the following type signature:

   ```
   filterM :: forall m a. Monad m => (a -> m Boolean) -> List a -> m (List a)
   ```

6. (Difficult) Every monad has a default `Functor` instance given by:

   ```
   map f a = do
     x <- a
     pure (f x)
   ```

   Use the monad laws to prove that for any monad, the following holds:

   ```
   lift2 f (pure a) (pure b) = pure (f a b)
   ```

   Where the `Apply` instance uses the `ap` function defined above. Recall that `lift2` was defined as follows:

   ```
   lift2 :: forall f a b c. Apply f => (a -> b -> c) -> f a -> f b -> f c
   lift2 f a b = f <$> a <*> b
   ```

   Note: There are no tests for this exercise.

Native Effects

We will now look at one particular monad of central importance in PureScript – the `Effect` monad.

The `Effect` monad is defined in the `Effect` module. It is used to manage so-called *native* side-effects. If you are familiar with Haskell, it is the equivalent of the `IO` monad.

What are native side-effects? They are the side-effects that distinguish JavaScript expressions from idiomatic PureScript expressions, which typically are free from side-effects. Some examples of native effects are:

- Console IO
- Random number generation

- Exceptions
- Reading/writing mutable state

And in the browser:

- DOM manipulation
- XMLHttpRequest / AJAX calls
- Interacting with a websocket
- Writing/reading to/from local storage

We have already seen plenty of examples of "non-native" side-effects:

- Optional values, as represented by the `Maybe` data type
- Errors, as represented by the `Either` data type
- Multi-functions, as represented by arrays or lists

Note that the distinction is subtle. It is true, for example, that an error message is a possible side-effect of a JavaScript expression in the form of an exception. In that sense, exceptions do represent native side-effects, and it is possible to represent them using `Effect`. However, error messages implemented using `Either` are not a side-effect of the JavaScript runtime, and so it is not appropriate to implement error messages in that style using `Effect`. So it is not the effect itself, which is native, but rather how it is implemented at runtime.

Side-Effects and Purity

In a pure language like PureScript, one question presents itself: without side-effects, how can one write useful real-world code?

The answer is that PureScript does not aim to eliminate side-effects but to represent them in such a way that pure computations can be distinguished from computations with side-effects in the type system. In this sense, the language is still pure.

Values with side-effects have different types from pure values. As such, it is impossible to pass a side-effecting argument to a function, for example, and have side-effects performed unexpectedly.

The only way side-effects managed by the `Effect` monad will be presented is to run a computation of type `Effect a` from JavaScript.

The Spago build tool (and other tools) provide a shortcut by generating additional JavaScript

to invoke the `main` computation when the application starts. `main` is required to be a computation in the `Effect` monad.

The Effect Monad

The `Effect` monad provides a well-typed API for computations with side-effects, while at the same time generating efficient JavaScript.

Let's look at the return type of the familiar `log` function. `Effect` indicates that this function produces a native effect, console IO in this case.

`Unit` indicates that no *meaningful* data is returned. You can think of `Unit` as analogous to the `void` keyword in other languages, such as C, Java, etc.

```
log :: String -> Effect Unit
```

> *Aside:* You may encounter IDE suggestions for the more general (and more elaborately typed) `log` function from `Effect.Class.Console`. This is interchangeable with the one from `Effect.Console` when dealing with the basic `Effect` monad. Reasons for the more general version will become clearer after reading about "Monad Transformers" in the "Monadic Adventures" chapter. For the curious (and impatient), this works because there's a `MonadEffect` instance for `Effect`.
>
> ```
> log :: forall m. MonadEffect m => String -> m Unit
> ```

Now let's consider an `Effect` that returns meaningful data. The `random` function from `Effect.Random` produces a random `Number`.

```
random :: Effect Number
```

Here's a full example program (found in `test/Random.purs` of this chapter's exercises folder).

```
module Test.Random where

import Prelude
import Effect (Effect)
import Effect.Random (random)
import Effect.Console (logShow)

main :: Effect Unit
main = do
  n <- random
  logShow n
```

Because `Effect` is a monad, we use do notation to *unwrap* the data it contains before passing this data on to the effectful `logShow` function. As a refresher, here's the equivalent code written using the `bind` operator:

```
main :: Effect Unit
main = random >>= logShow
```

Try running this yourself with:

```
spago run --main Test.Random
```

You should see a randomly chosen number between `0.0` and `1.0` printed to the console.

> *Aside:* `spago run` defaults to searching in the `Main` module for a `main` function. You may also specify an alternate module as an entry point with the `--main` flag, as in the above example. Just be sure that this alternate module also contains a `main` function.

Note that it's also possible to generate "random" (technically pseudorandom) data without resorting to impure effectful code. We'll cover these techniques in the "Generative Testing" chapter.

As mentioned previously, the `Effect` monad is of central importance to PureScript. The reason why it's central is that it is the conventional way to interoperate with PureScript's `Foreign Function Interface`, which provides the mechanism to execute a program and perform side effects. While it's desirable to avoid using the `Foreign Function Interface`, it's fairly critical to understand how it works and how to use it, so I recommend reading that chapter before doing any serious PureScript work. That said, the `Effect` monad is fairly simple. It has a few helper functions but doesn't do much except encapsulate side effects.

Exceptions

Let's examine a function from the `node-fs` package that involves two *native* side effects: reading mutable state and exceptions:

```
readTextFile :: Encoding -> String -> Effect String
```

If we attempt to read a file that does not exist:

```
import Node.Encoding (Encoding(..))
import Node.FS.Sync (readTextFile)

main :: Effect Unit
main = do
  lines <- readTextFile UTF8 "iDoNotExist.md"
  log lines
```

We encounter the following exception:

```
    throw err;
    ^
Error: ENOENT: no such file or directory, open 'iDoNotExist.md'
...
  errno: -2,
  syscall: 'open',
  code: 'ENOENT',
  path: 'iDoNotExist.md'
```

To manage this exception gracefully, we can wrap the potentially problematic code in `try` to handle either outcome:

```
main :: Effect Unit
main = do
  result <- try $ readTextFile UTF8 "iDoNotExist.md"
  case result of
    Right lines -> log $ "Contents: \n" <> lines
    Left  error -> log $ "Couldn't open file. Error was: " <> message error
```

`try` runs an `Effect` and returns eventual exceptions as a `Left` value. If the computation succeeds, the result gets wrapped in a `Right`:

```
try :: forall a. Effect a -> Effect (Either Error a)
```

We can also generate our own exceptions. Here is an alternative implementation of

`Data.List.head` that throws an exception if the list is empty rather than returning a `Maybe` value of `Nothing`.

```
exceptionHead :: List Int -> Effect Int
exceptionHead l = case l of
  x : _ -> pure x
  Nil -> throwException $ error "empty list"
```

Note that the `exceptionHead` function is a somewhat impractical example, as it is best to avoid generating exceptions in PureScript code and instead use non-native effects such as `Either` and `Maybe` to manage errors and missing values.

Mutable State

There is another effect defined in the core libraries: the `ST` effect.

The `ST` effect is used to manipulate mutable state. As pure functional programmers, we know that shared mutable state can be problematic. However, the `ST` effect uses the type system to restrict sharing in such a way that only safe *local* mutation is allowed.

The `ST` effect is defined in the `Control.Monad.ST` module. To see how it works, we need to look at the types of its actions:

```
new :: forall a r. a -> ST r (STRef r a)

read :: forall a r. STRef r a -> ST r a

write :: forall a r. a -> STRef r a -> ST r a

modify :: forall r a. (a -> a) -> STRef r a -> ST r a
```

`new` is used to create a new mutable reference cell of type `STRef r a`, which can be read using the `read` action and modified using the `write` and `modify` actions. The type `a` is the type of the value stored in the cell, and the type `r` is used to indicate a *memory region* (or *heap*) in the type system.

Here is an example. Suppose we want to simulate the movement of a particle falling under gravity by iterating a simple update function over many small time steps.

We can do this by creating a mutable reference cell to hold the position and velocity of the particle, and then using a `for` loop to update the value stored in that cell:

```
import Prelude

import Control.Monad.ST.Ref (modify, new, read)
import Control.Monad.ST (ST, for, run)

simulate :: forall r. Number -> Number -> Int -> ST r Number
simulate x0 v0 time = do
  ref <- new { x: x0, v: v0 }
  for 0 (time * 1000) \_ ->
    modify
      ( \o ->
          { v: o.v - 9.81 * 0.001
          , x: o.x + o.v * 0.001
          }
      )
      ref
  final <- read ref
  pure final.x
```

At the end of the computation, we read the final value of the reference cell and return the position of the particle.

Note that even though this function uses a mutable state, it is still a pure function, so long as the reference cell `ref` is not allowed to be used by other program parts. We will see that this is exactly what the `ST` effect disallows.

To run a computation with the `ST` effect, we have to use the `run` function:

```
run :: forall a. (forall r. ST r a) -> a
```

The thing to notice here is that the region type `r` is quantified *inside the parentheses* on the left of the function arrow. That means that whatever action we pass to `run` has to work with *any region* `r` whatsoever.

However, once a reference cell has been created by `new`, its region type is already fixed, so it would be a type error to try to use the reference cell outside the code delimited by `run`. This allows `run` to safely remove the `ST` effect and turn `simulate` into a pure function!

```
simulate' :: Number -> Number -> Int -> Number
simulate' x0 v0 time = run (simulate x0 v0 time)
```

You can even try running this function in PSCi:

```
> import Main

> simulate' 100.0 0.0 0
100.00

> simulate' 100.0 0.0 1
95.10

> simulate' 100.0 0.0 2
80.39

> simulate' 100.0 0.0 3
55.87

> simulate' 100.0 0.0 4
21.54
```

In fact, if we inline the definition of `simulate` at the call to `run`, as follows:

```
simulate :: Number -> Number -> Int -> Number
simulate x0 v0 time =
  run do
    ref <- new { x: x0, v: v0 }
    for 0 (time * 1000) \_ ->
      modify
        ( \o ->
            { v: o.v - 9.81 * 0.001
            , x: o.x + o.v * 0.001
            }
        )
        ref
    final <- read ref
    pure final.x
```

Then the compiler will notice that the reference cell cannot escape its scope and can safely turn `ref` into a `var`. Here is the generated JavaScript for `simulate` inlined with `run`:

```
var simulate = function (x0) {
  return function (v0) {
    return function (time) {
      return (function __do() {

        var ref = { value: { x: x0, v: v0 } };

        Control_Monad_ST_Internal["for"](0)(time * 1000 | 0)(function (v) {
          return Control_Monad_ST_Internal.modify(function (o) {
            return {
              v: o.v - 9.81 * 1.0e-3,
              x: o.x + o.v * 1.0e-3
            };
          })(ref);
        })();

        return ref.value.x;

      })();
    };
  };
};
```

Note that this resulting JavaScript is not as optimal as it could be. See this issue for more details. The above snippet should be updated once that issue is resolved.

For comparison, this is the generated JavaScript of the non-inlined form:

```
var simulate = function (x0) {
  return function (v0) {
    return function (time) {
      return function __do() {

        var ref = Control_Monad_ST_Internal["new"]({ x: x0, v: v0 })();

        Control_Monad_ST_Internal["for"](0)(time * 1000 | 0)(function (v) {
          return Control_Monad_ST_Internal.modify(function (o) {
            return {
              v: o.v - 9.81 * 1.0e-3,
              x: o.x + o.v * 1.0e-3
            };
          })(ref);
        })();

        var $$final = Control_Monad_ST_Internal.read(ref)();
        return $$final.x;
      };
    };
  };
};
```

The `ST` effect is a good way to generate short JavaScript when working with locally-scoped mutable state, especially when used together with actions like `for`, `foreach`, and `while`, which generate efficient loops.

Exercises

1. (Medium) Rewrite the `safeDivide` function as `exceptionDivide` and throw an exception using `throwException` with the message `"div zero"` if the denominator is zero.
2. (Medium) Write a function `estimatePi :: Int -> Number` that uses `n` terms of the Gregory Series to calculate an approximation of `pi`. *Hints:* You can pattern your answer like the definition of `simulate` above. You might need to convert an `Int` into a `Number` using `toNumber :: Int -> Number` from `Data.Int`.
3. (Medium) Write a function `fibonacci :: Int -> Int` to compute the `n` th Fibonacci number, using `ST` to track the values of the previous two Fibonacci numbers. Using PSCi, compare the speed of your new `ST`-based implementation against the recursive implementation (`fib`) from Chapter 5.

DOM Effects

In the final sections of this chapter, we will apply what we have learned about effects in the `Effect` monad to the problem of working with the DOM.

There are several PureScript packages for working directly with the DOM or open-source DOM libraries. For example:

- `web-dom` provides type definitions and low-level interface implementations for the W3C DOM spec.
- `web-html` provides type definitions and low-level interface implementations for the W3C HTML5 spec.
- `jquery` is a set of bindings to the jQuery library.

There are also PureScript libraries that build abstractions on top of these libraries, such as

- `thermite` builds on `react`
- `react-basic-hooks` builds on `react-basic`
- `halogen` provides a type-safe set of abstractions on top of a custom virtual DOM library.

In this chapter, we will use the `react-basic-hooks` library to add a user interface to our address book application, but the interested reader is encouraged to explore alternative approaches.

An Address Book User Interface

Using the `react-basic-hooks` library, we will define our application as a React *component*. React components describe HTML elements in code as pure data structures, which are then efficiently rendered to the DOM. In addition, components can respond to events like button clicks. The `react-basic-hooks` library uses the `Effect` monad to describe how to handle these events.

A full tutorial for the React library is well beyond the scope of this chapter, but the reader is encouraged to consult its documentation where needed. For our purposes, React will provide a practical example of the `Effect` monad.

We are going to build a form that will allow a user to add a new entry into our address book. The form will contain text boxes for the various fields (first name, last name, city, state, etc.) and an area where validation errors will be displayed. As the user types text into the text

boxes, the validation errors will be updated.

To keep things simple, the form will have a fixed shape: the different phone number types (home, cell, work, other) will be expanded into separate text boxes.

You can launch the web app from the `exercises/chapter8` directory with the following commands:

```
$ npm install
$ npx spago build
$ npx parcel src/index.html --open
```

If development tools such as `spago` and `parcel` are installed globally, then the `npx` prefix may be omitted. You have likely already installed `spago` globally with `npm i -g spago`, and the same can be done for `parcel`.

`parcel` should launch a browser window with our "Address Book" app. If you keep the `parcel` terminal open and rebuild with `spago` in another terminal, the page should automatically refresh with your latest edits. You can also configure automatic rebuilds (and therefore automatic page refresh) on file-save if you're using an editor that supports `purs ide` or are running `pscid`.

In this Address Book app, you can enter some values into the form fields and see the validation errors printed onto the page.

Let's explore how it works.

The `src/index.html` file is minimal:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Address Book</title>
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/
bootstrap/4.4.1/css/bootstrap.min.css" crossorigin="anonymous">
  </head>
  <body>
    <div id="container"></div>
    <script type="module" src="./index.js"></script>
  </body>
</html>
```

The `<script` line includes the JavaScript entry point, `index.js`, which contains this single line:

```
import { main } from "../output/Main/index.js";

main();
```

It calls our generated JavaScript equivalent of the `main` function of `module Main` (`src/main.purs`). Recall that `spago build` puts all generated JavaScript in the `output` directory.

The `main` function uses the DOM and HTML APIs to render our address book component within the `container` element we defined in `index.html`:

```
main :: Effect Unit
main = do
  log "Rendering address book component"
  -- Get window object
  w <- window
  -- Get window's HTML document
  doc <- document w
  -- Get "container" element in HTML
  ctr <- getElementById "container" $ toNonElementParentNode doc
  case ctr of
    Nothing -> throw "Container element not found."
    Just c -> do
      -- Create AddressBook react component
      addressBookApp <- mkAddressBookApp
      let
        -- Create JSX node from react component. Pass-in empty props
        app = element addressBookApp {}
      -- Render AddressBook JSX node in DOM "container" element
      D.render app c
```

Note that these three lines:

```
w <- window
doc <- document w
ctr <- getElementById "container" $ toNonElementParentNode doc
```

Can be consolidated to:

```
doc <- document =<< window
ctr <- getElementById "container" $ toNonElementParentNode doc
```

Or consolidated even further to:

```
ctr <- getElementById "container" <<< toNonElementParentNode =<< document =<<
window
-- or, equivalently:
ctr <- window >>= document >>= toNonElementParentNode >>> getElementById
"container"
```

It is a matter of personal preference whether the intermediate `w` and `doc` variables aid in readability.

Let's dig into our AddressBook `reactComponent`. We'll start with a simplified component and then build up to the actual code in `Main.purs`.

Take a look at this minimal component. Feel free to substitute the full component with this one to see it run:

```
mkAddressBookApp :: Effect (ReactComponent {})
mkAddressBookApp =
  reactComponent
    "AddressBookApp"
    (\props -> pure $ D.text "Hi! I'm an address book")
```

`reactComponent` has this intimidating signature:

```
reactComponent ::
  forall hooks props.
  Lacks "children" props =>
  Lacks "key" props =>
  Lacks "ref" props =>
  String ->
  ({ | props } -> Render Unit hooks JSX) ->
  Effect (ReactComponent { | props })
```

The important points to note are the arguments after all the type class constraints. It takes a `String` (an arbitrary component name), a function that describes how to convert `props` into rendered `JSX`, and returns our `ReactComponent` wrapped in an `Effect`.

The props-to-JSX function is simply:

```
\props -> pure $ D.text "Hi! I'm an address book"
```

`props` are ignored, `D.text` returns `JSX`, and `pure` lifts to rendered JSX. Now `component` has everything it needs to produce the `ReactComponent`.

Next, we'll examine some of the additional complexities of the full Address Book component.

These are the first few lines of our full component:

```
mkAddressBookApp :: Effect (ReactComponent {})
mkAddressBookApp = do
  reactComponent "AddressBookApp" \props -> R.do
    Tuple person setPerson <- useState examplePerson
```

We track `person` as a piece of state with the `useState` hook.

```
Tuple person setPerson <- useState examplePerson
```

Note that you are free to break-up component state into multiple pieces of state with multiple calls to `useState`. For example, we could rewrite this app to use a separate piece of state for each record field of `Person`, but that results in a slightly less convenient architecture in this case.

In other examples, you may encounter the `/\` infix operator for `Tuple`. This is equivalent to the above line:

```
firstName /\ setFirstName <- useState p.firstName
```

`useState` takes a default initial value and returns the current value and a way to update the value. We can check the type of `useState` to gain more insight of the types `person` and `setPerson`:

```
useState ::
  forall state.
  state ->
  Hook (UseState state) (Tuple state ((state -> state) -> Effect Unit))
```

We can strip the `Hook (UseState state)` wrapper off of the return value because `useState` is called within an `R.do` block. We'll elaborate on `R.do` later.

So now we can observe the following signatures:

```
person :: state
setPerson :: (state -> state) -> Effect Unit
```

The specific type of `state` is determined by our initial default value. `Person Record` in this case because that is the type of `examplePerson`.

`person` is how we access the current state at each rerender.

`setPerson` is how we update the state. We provide a function describing how to transform the current state into the new one. The record update syntax is perfect for this when the type of `state` happens to be a `Record`, for example:

```
setPerson (\currentPerson -> currentPerson {firstName = "NewName"})
```

Or as shorthand:

```
setPerson _ {firstName = "NewName"}
```

Non-`Record` states can also follow this update pattern. See this guide for more details on best practices.

Recall that `useState` is used within an `R.do` block. `R.do` is a special react hooks variant of `do`. The `R.` prefix "qualifies" this as coming from `React.Basic.Hooks`, and means we use their hooks-compatible version of `bind` in the `R.do` block. This is known as a "qualified do". It lets us ignore the `Hook (UseState state)` wrapping and bind the inner `Tuple` of values to variables.

Another possible state management strategy is with `useReducer`, but that is outside the scope of this chapter.

Rendering `JSX` occurs here:

```
  pure
    $ D.div
        { className: "container"
        , children:
            renderValidationErrors errors
              <> [ D.div
                     { className: "row"
                     , children:
                         [ D.form_
                             $ [ D.h3_ [ D.text "Basic Information" ]
                               , formField "First Name" "First Name"
person.firstName \s ->
                                   setPerson _ { firstName = s }
                               , formField "Last Name" "Last Name" person.lastName
\s ->
                                   setPerson _ { lastName = s }
                               , D.h3_ [ D.text "Address" ]
                               , formField "Street" "Street"
person.homeAddress.street \s ->
                                   setPerson _ { homeAddress { street = s } }
                               , formField "City" "City" person.homeAddress.city
\s ->
                                   setPerson _ { homeAddress { city = s } }
                               , formField "State" "State"
person.homeAddress.state \s ->
                                   setPerson _ { homeAddress { state = s } }
                               , D.h3_ [ D.text "Contact Information" ]
                               ]
                             <> renderPhoneNumbers
                         ]
                     }
                 ]
        }
```

Here we produce `JSX`, which represents the intended state of the DOM. This JSX is typically created by applying functions corresponding to HTML tags (e.g., `div`, `form`, `h3`, `li`, `ul`, `label`, `input`) which create single HTML elements. These HTML elements are React components themselves, converted to JSX. There are usually three variants of each of these functions:

- `div_`: Accepts an array of child elements. Uses default attributes.
- `div`: Accepts a `Record` of attributes. An array of child elements may be passed to the `children` field of this record.
- `div'`: Same as `div`, but returns the `ReactComponent` before conversion to `JSX`.

To display validation errors (if any) at the top of our form, we create a `renderValidationErrors` helper function that turns the `Errors` structure into an array of JSX. This array is prepended to the rest of our form.

```
renderValidationErrors :: Errors -> Array R.JSX
renderValidationErrors [] = []
renderValidationErrors xs =
  let
    renderError :: String -> R.JSX
    renderError err = D.li_ [ D.text err ]
  in
    [ D.div
        { className: "alert alert-danger row"
        , children: [ D.ul_ (map renderError xs) ]
        }
    ]
```

Note that since we are simply manipulating regular data structures here, we can use functions like `map` to build up more interesting elements:

```
children: [ D.ul_ (map renderError xs)]
```

We use the `className` property to define classes for CSS styling. We're using the Bootstrap `stylesheet` for this project, which is imported in `index.html`. For example, we want items in our form arranged as `row`s, and validation errors to be emphasized with `alert-danger` styling:

```
className: "alert alert-danger row"
```

A second helper function is `formField`, which creates a text input for a single form field:

```
formField :: String -> String -> String -> (String -> Effect Unit) -> R.JSX
formField name placeholder value setValue =
  D.div
    { className: "form-group row"
    , children:
        [ D.label
            { className: "col-sm col-form-label"
            , htmlFor: name
            , children: [ D.text name ]
            }
        , D.div
            { className: "col-sm"
            , children:
                [ D.input
                    { className: "form-control"
                    , id: name
                    , placeholder
                    , value
                    , onChange:
                        let
                          handleValue :: Maybe String -> Effect Unit
                          handleValue (Just v) = setValue v
                          handleValue Nothing  = pure unit
                        in
                          handler targetValue handleValue
                    }
                ]
            }
        ]
    }
```

Putting the `input` and display `text` in a `label` aids in accessibility for screen readers.

The `onChange` attribute allows us to describe how to respond to user input. We use the `handler` function, which has the following type:

```
handler :: forall a. EventFn SyntheticEvent a -> (a -> Effect Unit) ->
EventHandler
```

For the first argument to `handler` we use `targetValue`, which provides the value of the text within the HTML `input` element. It matches the signature expected by `handler` where the type variable `a` in this case is `Maybe String`:

```
targetValue :: EventFn SyntheticEvent (Maybe String)
```

In JavaScript, the `input` element's `onChange` event is accompanied by a `String` value, but since strings in JavaScript can be null, `Maybe` is used for safety.

The second argument to `handler`, `(a -> Effect Unit)`, must therefore have this signature:

```
Maybe String -> Effect Unit
```

It is a function that describes how to convert this `Maybe String` value into our desired effect. We define a custom `handleValue` function for this purpose and pass it to `handler` as follows:

```
onChange:
  let
    handleValue :: Maybe String -> Effect Unit
    handleValue (Just v) = setValue v
    handleValue Nothing  = pure unit
  in
    handler targetValue handleValue
```

`setValue` is the function we provided to each `formField` call that takes a string and makes the appropriate record-update call to the `setPerson` hook.

Note that `handleValue` can be substituted as:

```
onChange: handler targetValue $ traverse_ setValue
```

Feel free to investigate the definition of `traverse_` to see how both forms are indeed equivalent.

That covers the basics of our component implementation. However, you should read the source accompanying this chapter to get a full understanding of the way the component works.

Obviously, this user interface can be improved in a number of ways. The exercises will explore some ways in which we can make the application more usable.

Exercises

Modify `src/Main.purs` in the following exercises. There are no unit tests for these exercises.

1. (Easy) Modify the application to include a work phone number text box.
2. (Medium) Right now, the application shows validation errors collected in a single "pink-

alert" background. Modify to give each validation error its own pink-alert background by separating them with blank lines.

Hint: Instead of using a `ul` element to show the validation errors in a list, modify the code to create one `div` with the `alert` and `alert-danger` styles for each error.

3. (Difficult, Extended) One problem with this user interface is that the validation errors are not displayed next to the form fields they originated from. Modify the code to fix this problem.

   Hint: The error type returned by the validator should be extended to indicate which field caused the error. You might want to use the following modified `Errors` type:

   ```
   data Field = FirstNameField
              | LastNameField
              | StreetField
              | CityField
              | StateField
              | PhoneField PhoneType

   data ValidationError = ValidationError String Field

   type Errors = Array ValidationError
   ```

   You will need to write a function that extracts the validation error for a particular `Field` from the `Errors` structure.

Conclusion

This chapter has covered a lot of ideas about handling side-effects in PureScript:

- We met the `Monad` type class and its connection to do notation.
- We introduced the monad laws and saw how they allow us to transform code written using do notation.
- We saw how monads can be used abstractly to write code that works with different side-effects.
- We saw how monads are examples of applicative functors, how both allow us to compute with side-effects, and the differences between the two approaches.
- The concept of native effects was defined, and we met the `Effect` monad, which handles native side-effects.

- We used the `Effect` monad to handle a variety of effects: random number generation, exceptions, console IO, mutable state, and DOM manipulation using React.

The `Effect` monad is a fundamental tool in real-world PureScript code. It will be used in the rest of the book to handle side-effects in a number of other use-cases.

Asynchronous Effects

Chapter Goals

This chapter focuses on the `Aff` monad, which is similar to the `Effect` monad, but represents *asynchronous* side-effects. We'll demonstrate examples of asynchronously interacting with the filesystem and making HTTP requests. We'll also cover managing sequential and parallel execution of asynchronous effects.

Project Setup

New PureScript libraries introduced in this chapter are:

- `aff` - defines the `Aff` monad.
- `node-fs-aff` - asynchronous filesystem operations with `Aff`.
- `affjax` - HTTP requests with AJAX and `Aff`.
- `parallel` - parallel execution of `Aff`.

When running outside of the browser (such as in our Node.js environment), the `affjax` library requires the `xhr2` NPM module, which is listed as a dependency in the `package.json` of this chapter. Install that by running:

```
$ npm install
```

Asynchronous JavaScript

A convenient way to work with asynchronous code in JavaScript is with `async` and `await`. See this article on asynchronous JavaScript for more background information.

Here is an example of using this technique to copy the contents of one file to another file:

```
import { promises as fsPromises } from 'fs'

async function copyFile(file1, file2) {
  let data = await fsPromises.readFile(file1, { encoding: 'utf-8' });
  fsPromises.writeFile(file2, data, { encoding: 'utf-8' });
}

copyFile('file1.txt', 'file2.txt')
.catch(e => {
  console.log('There was a problem with copyFile: ' + e.message);
});
```

It is also possible to use callbacks or synchronous functions, but those are less desirable because:

- Callbacks lead to excessive nesting, known as "Callback Hell" or the "Pyramid of Doom".
- Synchronous functions block execution of the other code in your app.

Asynchronous PureScript

The `Aff` monad in PureScript offers similar ergonomics of JavaScript's `async` / `await` syntax. Here is the same `copyFile` example from before, but rewritten in PureScript using `Aff`:

```
import Prelude
import Data.Either (Either(..))
import Effect.Aff (Aff, attempt, message, launchAff_)
import Effect (Effect)
import Effect.Class.Console (log)
import Node.Encoding (Encoding(..))
import Node.FS.Aff (readTextFile, writeTextFile)
import Node.Path (FilePath)

main :: Effect Unit
main = launchAff_ program

program :: Aff Unit
program = do
  result <- attempt $ copyFile "file1.txt" "file2.txt"
  case result of
    Left e -> log $ "There was a problem with copyFile: " <> message e
    _ -> pure unit

copyFile :: FilePath -> FilePath -> Aff Unit
copyFile file1 file2 = do
  my_data <- readTextFile UTF8 file1
  writeTextFile UTF8 file2 my_data
```

Note that we have to use `launchAff_` to convert the `Aff` to `Effect` because `main` must be `Effect Unit`.

It is also possible to re-write the above snippet using callbacks or synchronous functions (for example, with `Node.FS.Async` and `Node.FS.Sync`, respectively), but those share the same downsides as discussed earlier with JavaScript, so that coding style is not recommended.

The syntax for working with `Aff` is very similar to working with `Effect`. They are both monads and can therefore be written with do notation.

For example, if we look at the signature of `readTextFile`, we see that it returns the file contents as a `String` wrapped in `Aff`:

```
readTextFile :: Encoding -> FilePath -> Aff String
```

We can "unwrap" the returned string with a bind arrow (`<-`) in do notation:

```
my_data <- readTextFile UTF8 file1
```

Then pass it as the string argument to `writeTextFile`:

```
writeTextFile :: Encoding -> FilePath -> String -> Aff Unit
```

The only other notable feature unique to `Aff` in the above example is `attempt`, which captures errors or exceptions encountered while running `Aff` code and stores them in an `Either`:

```
attempt :: forall a. Aff a -> Aff (Either Error a)
```

You should hopefully be able to draw on your knowledge of concepts from previous chapters and combine this with the new `Aff` patterns learned in the above `copyFile` example to tackle the following exercises:

Exercises

1. (Easy) Write a `concatenateFiles` function that concatenates two text files.
2. (Medium) Write a function `concatenateMany` to concatenate multiple text files, given an array of input and output file names. *Hint*: use `traverse`.
3. (Medium) Write a function `countCharacters :: FilePath -> Aff (Either Error Int)` that returns the number of characters in a file, or an error if one is encountered.

Additional Aff Resources

If you haven't already looked at the official Aff guide, skim through that now. It's not a direct prerequisite for completing the remaining exercises in this chapter, but you may find it helpful to lookup some functions on Pursuit.

You're also welcome to consult these supplemental resources too, but again, the exercises in this chapter don't depend on them:

- Drew's Aff Post
- Additional Aff Explanation and Examples

A HTTP Client

The `affjax` library offers a convenient way to make asynchronous AJAX HTTP requests with `Aff`. Depending on what environment you are targeting, you need to use either the

purescript-affjax-web or the purescript-affjax-node library.

In the rest of this chapter, we will be targeting node and thus using `purescript-affjax-node`. Consult the Affjax docs for more usage information. Here is an example that makes HTTP GET requests at a provided URL and returns the response body or an error message:

```
import Prelude
import Affjax.Node as AN
import Affjax.ResponseFormat as ResponseFormat
import Data.Either (Either(..))
import Effect.Aff (Aff)

getUrl :: String -> Aff String
getUrl url = do
  result <- AN.get ResponseFormat.string url
  pure case result of
    Left err -> "GET /api response failed to decode: " <> AN.printError err
    Right response -> response.body
```

When calling this in the repl, `launchAff_` is required to convert the `Aff` to a repl-compatible `Effect`:

```
$ spago repl

> :pa
… import Prelude
… import Effect.Aff (launchAff_)
… import Effect.Class.Console (log)
… import Test.HTTP (getUrl)
…
… launchAff_ do
…   str <- getUrl "https://reqres.in/api/users/1"
…   log str
…
unit
{"data":
{"id":1,"email":"george.bluth@reqres.in","first_name":"George","last_name":"Blu
th", ...}}
```

Exercises

1. (Easy) Write a function `writeGet` which makes an HTTP `GET` request to a provided url, and writes the response body to a file.

Parallel Computations

We've seen how to use the `Aff` monad and do notation to compose asynchronous computations in sequence. It would also be useful to be able to compose asynchronous computations *in parallel*. With `Aff`, we can compute in parallel simply by initiating our two computations one after the other.

The `parallel` package defines a type class `Parallel` for monads like `Aff`, which support parallel execution. When we met applicative functors earlier in the book, we observed how applicative functors can be useful for combining parallel computations. In fact, an instance for `Parallel` defines a correspondence between a monad `m` (such as `Aff`) and an applicative functor `f` that can be used to combine computations in parallel:

```
class (Monad m, Applicative f) <= Parallel f m | m -> f, f -> m where
  sequential :: forall a. f a -> m a
  parallel :: forall a. m a -> f a
```

The class defines two functions:

- `parallel`, which takes computations in the monad `m` and turns them into computations in the applicative functor `f`, and
- `sequential`, which performs a conversion in the opposite direction.

The `aff` library provides a `Parallel` instance for the `Aff` monad. It uses mutable references to combine `Aff` actions in parallel by keeping track of which of the two continuations has been called. When both results have been returned, we can compute the final result and pass it to the main continuation.

Because applicative functors support lifting of functions of arbitrary arity, we can perform more computations in parallel by using the applicative combinators. We can also benefit from all of the standard library functions which work with applicative functors, such as `traverse` and `sequence`!

We can also combine parallel computations with sequential portions of code by using applicative combinators in a do notation block, or vice versa, using `parallel` and `sequential` to change type constructors where appropriate.

To demonstrate the difference between sequential and parallel execution, we'll create an array of 100 10-millisecond delays, then execute those delays with both techniques. You'll notice in the repl that `seqDelay` is much slower than `parDelay`. Note that parallel execution is enabled by simply replacing `sequence_` with `parSequence_`.

```
import Prelude

import Control.Parallel (parSequence_)
import Data.Array (replicate)
import Data.Foldable (sequence_)
import Effect (Effect)
import Effect.Aff (Aff, Milliseconds(..), delay, launchAff_)

delayArray :: Array (Aff Unit)
delayArray = replicate 100 $ delay $ Milliseconds 10.0

seqDelay :: Effect Unit
seqDelay = launchAff_ $ sequence_ delayArray

parDelay :: Effect Unit
parDelay = launchAff_ $ parSequence_ delayArray
```

```
$ spago repl

> import Test.ParallelDelay

> seqDelay -- This is slow
unit

> parDelay -- This is fast
unit
```

Here's a more real-world example of making multiple HTTP requests in parallel. We're reusing our `getUrl` function to fetch information from two users in parallel. Note that `parTraverse` (the parallel version of `traverse`) is used in this case. This example would also work fine with `traverse` instead, but it will be slower.

```
import Prelude

import Control.Parallel (parTraverse)
import Effect (Effect)
import Effect.Aff (launchAff_)
import Effect.Class.Console (logShow)
import Test.HTTP (getUrl)

fetchPar :: Effect Unit
fetchPar =
  launchAff_ do
    let
      urls = map (\n -> "https://reqres.in/api/users/" <> show n) [ 1, 2 ]
    res <- parTraverse getUrl urls
    logShow res
```

```
$ spago repl

> import Test.ParallelFetch

> fetchPar
unit
["{\"data\":{\"id\":1,\"email\":\"george.bluth@reqres.in\", ... }"
,"{\"data\":{\"id\":2,\"email\":\"janet.weaver@reqres.in\", ... }"
]
```

A full listing of available parallel functions can be found in the `parallel` docs on Pursuit. The aff docs section on parallel also contains more examples.

Exercises

1. (Easy) Write a `concatenateManyParallel` function with the same signature as the earlier `concatenateMany` function but reads all input files in parallel.
2. (Medium) Write a `getWithTimeout :: Number -> String -> Aff (Maybe String)` function which makes an HTTP `GET` request at the provided URL and returns either:
   - `Nothing` : if the request takes longer than the provided timeout (in milliseconds).
   - The string response: if the request succeeds before the timeout elapses.
3. (Difficult) Write a `recurseFiles` function that takes a "root" file and returns an array of all paths listed in that file (and listed in the listed files too). Read listed files in parallel. Paths are relative to the directory of the file they appear in. *Hint:* The `node-path` module has some helpful functions for negotiating directories.

For example, if starting from the following `root.txt` file:

```
$ cat root.txt
a.txt
b/a.txt
c/a/a.txt

$ cat a.txt
b/b.txt

$ cat b/b.txt
c/a.txt

$ cat b/c/a.txt

$ cat b/a.txt

$ cat c/a/a.txt
```

The expected output is:

```
["root.txt","a.txt","b/a.txt","b/b.txt","b/c/a.txt","c/a/a.txt"]
```

Conclusion

In this chapter, we covered asynchronous effects and learned how to:

- Run asynchronous code in the `Aff` monad with the `aff` library.
- Make HTTP requests asynchronously with the `affjax` library.
- Run asynchronous code in parallel with the `parallel` library.

The Foreign Function Interface

Chapter Goals

This chapter will introduce PureScript's *foreign function interface* (or *FFI*), which enables communication from PureScript code to JavaScript code and vice versa. We will cover how to:

- Call pure, effectful, and asynchronous JavaScript functions from PureScript.
- Work with untyped data.
- Encode and parse JSON using the `argonaut` package.

Towards the end of this chapter, we will revisit our recurring address book example. The goal of the chapter will be to add the following new functionality to our application using the FFI:

- Alert the user with a popup notification.
- Store the serialized form data in the browser's local storage, and reload it when the application restarts.

There is also an addendum covering some additional topics that are not as commonly sought-after. Feel free to read these sections, but don't let them stand in the way of progressing through the remainder of the book if they're less relevant to your learning objectives:

- Understand the representation of PureScript values at runtime.
- Call PureScript functions from JavaScript.

Project Setup

The source code for this module is a continuation of the source code from chapters 3, 7, and 8. As such, the source tree includes the appropriate source files from those chapters.

This chapter introduces the `argonaut` library as a dependency. This library is used for encoding and decoding JSON.

The exercises for this chapter should be written in `test/MySolutions.purs` and can be checked against the unit tests in `test/Main.purs` by running `spago test`.

The Address Book app can be launched with `parcel src/index.html --open`. It uses the same workflow from Chapter 8, so refer to that chapter for more detailed instructions.

A Disclaimer

PureScript provides a straightforward foreign function interface to make working with JavaScript as simple as possible. However, it should be noted that the FFI is an *advanced* feature of the language. To use it safely and effectively, you should understand the runtime representation of the data you plan to work with. This chapter aims to impart such an understanding as pertains to code in PureScript's standard libraries.

PureScript's FFI is designed to be very flexible. In practice, this means that developers have a choice between giving their foreign functions very simple types or using the type system to protect against accidental misuses of foreign code. Code in the standard libraries tends to favor the latter approach.

As a simple example, a JavaScript function makes no guarantees that its return value will not be `null`. Indeed, idiomatic JavaScript code returns `null` quite frequently! However, PureScript's types are usually not inhabited by a null value. Therefore, it is the responsibility of the developer to handle these corner cases appropriately when designing their interfaces to JavaScript code using the FFI.

Calling JavaScript From PureScript

The simplest way to use JavaScript code from PureScript is to give a type to an existing JavaScript value using a *foreign import* declaration. Foreign import declarations must have a corresponding JavaScript declaration *exported* from a *foreign JavaScript module*.

For example, consider the `encodeURIComponent` function, which can be used in JavaScript to encode a component of a URI by escaping special characters:

```
$ node

node> encodeURIComponent('Hello World')
'Hello%20World'
```

This function has the correct runtime representation for the function type `String -> String`, since it takes non-null strings to non-null strings and has no other side-effects.

We can assign this type to the function with the following foreign import declaration:

```
module Test.URI where

foreign import _encodeURIComponent :: String -> String
```

We also need to write a foreign JavaScript module to import it from. A corresponding foreign JavaScript module is one of the same name but the extension changed from `.purs` to `.js`. If the PureScript module above is saved as `URI.purs`, then the foreign JavaScript module is saved as `URI.js`. Since `encodeURIComponent` is already defined, we have to export it as `_encodeURIComponent`:

```
"use strict";

export const _encodeURIComponent = encodeURIComponent;
```

Since version 0.15, PureScript uses the ES module system when interoperating with JavaScript. In ES modules, functions and values are exported from a module by providing the `export` keyword on an object.

With these two pieces in place, we can now use the `_encodeURIComponent` function from PureScript like any function written in PureScript. For example, in PSCi, we can reproduce the calculation above:

```
$ spago repl

> import Test.URI
> _encodeURIComponent "Hello World"
"Hello%20World"
```

We can also define our own functions in foreign modules. Here's an example of how to create and call a custom JavaScript function that squares a `Number`:

`test/Examples.js`:

```
"use strict";

export const square = function (n) {
  return n * n;
};
```

`test/Examples.purs`:

```
module Test.Examples where

foreign import square :: Number -> Number
```

```
$ spago repl

> import Test.Examples
> square 5.0
25.0
```

Functions of Multiple Arguments

Let's rewrite our `diagonal` function from Chapter 2 in a foreign module. This function calculates the diagonal of a right-angled triangle.

```
foreign import diagonal :: Number -> Number -> Number
```

Recall that functions in PureScript are *curried*. `diagonal` is a function that takes a `Number` and returns a *function* that takes a `Number` and returns a `Number`.

```
export const diagonal = function (w) {
  return function (h) {
    return Math.sqrt(w * w + h * h);
  };
};
```

Or with ES6 arrow syntax (see ES6 note below).

```
export const diagonalArrow = w => h =>
  Math.sqrt(w * w + h * h);
```

```
foreign import diagonalArrow :: Number -> Number -> Number
```

```
$ spago repl

> import Test.Examples
> diagonal 3.0 4.0
5.0
> diagonalArrow 3.0 4.0
5.0
```

Uncurried Functions

Writing curried functions in JavaScript isn't always feasible, despite being scarcely idiomatic. A typical multi-argument JavaScript function would be of the *uncurried* form:

```
export const diagonalUncurried = function (w, h) {
  return Math.sqrt(w * w + h * h);
};
```

The module `Data.Function.Uncurried` exports *wrapper* types and utility functions to work with uncurried functions.

```
foreign import diagonalUncurried :: Fn2 Number Number Number
```

Inspecting the type constructor `Fn2`:

```
$ spago repl

> import Data.Function.Uncurried
> :kind Fn2
Type -> Type -> Type -> Type
```

`Fn2` takes three type arguments. `Fn2 a b c` is a type representing an uncurried function of two arguments of types `a` and `b`, that returns a value of type `c`. We used it to import `diagonalUncurried` from the foreign module.

We can then call it with `runFn2`, which takes the uncurried function and then the arguments.

```
$ spago repl

> import Test.Examples
> import Data.Function.Uncurried
> runFn2 diagonalUncurried 3.0 4.0
5.0
```

The `functions` package defines similar type constructors for function arities from 0 to 10.

A Note About Uncurried Functions

PureScript's curried functions have certain advantages. It allows us to partially apply

functions, and to give type class instances for function types – but it comes with a performance penalty. For performance-critical code, it is sometimes necessary to define uncurried JavaScript functions which accept multiple arguments.

We can also create uncurried functions from PureScript. For a function of two arguments, we can use the `mkFn2` function.

```
uncurriedAdd :: Fn2 Int Int Int
uncurriedAdd = mkFn2 \n m -> m + n
```

We can apply the uncurried function of two arguments by using `runFn2` as before:

```
uncurriedSum :: Int
uncurriedSum = runFn2 uncurriedAdd 3 10
```

The key here is that the compiler *inlines* the `mkFn2` and `runFn2` functions whenever they are fully applied. The result is that the generated code is very compact:

```
var uncurriedAdd = function (n, m) {
  return m + n | 0;
};

var uncurriedSum = uncurriedAdd(3, 10);
```

For contrast, here is a traditional curried function:

```
curriedAdd :: Int -> Int -> Int
curriedAdd n m = m + n

curriedSum :: Int
curriedSum = curriedAdd 3 10
```

And the resulting generated code, which is less compact due to the nested functions:

```
var curriedAdd = function (n) {
  return function (m) {
    return m + n | 0;
  };
};

var curriedSum = curriedAdd(3)(10);
```

A Note About Modern JavaScript Syntax

The arrow function syntax we saw earlier is an ES6 feature, which is incompatible with some older browsers (namely IE11). As of writing, it is estimated that arrow functions are unavailable for the 6% of users who have not yet updated their web browser.

To be compatible with the most users, the JavaScript code generated by the PureScript compiler does not use arrow functions. It is also recommended to **avoid arrow functions in public libraries** for the same reason.

You may still use arrow functions in your own FFI code, but then you should include a tool such as Babel in your deployment workflow to convert these back to ES5 compatible functions.

If you find arrow functions in ES6 more readable, you may transform JavaScript code in the compiler's `output` directory with a tool like Lebab:

```
npm i -g lebab
lebab --replace output/ --transform arrow,arrow-return
```

This operation would convert the above `curriedAdd` function to:

```
var curriedAdd = n => m =>
  m + n | 0;
```

The remaining examples in this book will use arrow functions instead of nested functions.

Exercises

1. (Medium) Write a JavaScript function `volumeFn` in the `Test.MySolutions` module that finds the volume of a box. Use an `Fn` wrapper from `Data.Function.Uncurried`.
2. (Medium) Rewrite `volumeFn` with arrow functions as `volumeArrow`.

Passing Simple Types

The following data types may be passed between PureScript and JavaScript as-is:

PureScript	JavaScript

PureScript	JavaScript
Boolean	Boolean
String	String
Int, Number	Number
Array	Array
Record	Object

We've already seen examples with the primitive types `String` and `Number`. We'll now take a look at the structural types `Array` and `Record` (`Object` in JavaScript).

To demonstrate passing `Array`s, here's how to call a JavaScript function that takes an `Array` of `Int` and returns the cumulative sum as another array. Recall that since JavaScript does not have a separate type for `Int`, both `Int` and `Number` in PureScript translate to `Number` in JavaScript.

```
foreign import cumulativeSums :: Array Int -> Array Int
```

```
export const cumulativeSums = arr => {
  let sum = 0
  let sums = []
  arr.forEach(x => {
    sum += x;
    sums.push(sum);
  });
  return sums;
};
```

```
$ spago repl

> import Test.Examples
> cumulativeSums [1, 2, 3]
[1,3,6]
```

To demonstrate passing `Records`, here's how to call a JavaScript function that takes two `Complex` numbers as records and returns their sum as another record. Note that a `Record` in PureScript is represented as an `Object` in JavaScript:

```
type Complex = {
  real :: Number,
  imag :: Number
}

foreign import addComplex :: Complex -> Complex -> Complex
```

```
export const addComplex = a => b => {
  return {
    real: a.real + b.real,
    imag: a.imag + b.imag
  }
};
```

```
$ spago repl

> import Test.Examples
> addComplex { real: 1.0, imag: 2.0 } { real: 3.0, imag: 4.0 }
{ imag: 6.0, real: 4.0 }
```

Note that the above techniques require trusting that JavaScript will return the expected types, as PureScript cannot apply type checking to JavaScript code. We will describe this type safety concern in more detail later on in the JSON section, as well as cover techniques to protect against type mismatches.

Exercises

1. (Medium) Write a JavaScript function `cumulativeSumsComplex` (and corresponding PureScript foreign import) that takes an `Array` of `Complex` numbers and returns the cumulative sum as another array of complex numbers.

Beyond Simple Types

We have seen examples of how to send and receive types with a native JavaScript representation, such as `String`, `Number`, `Array`, and `Record`, over FFI. Now we'll cover how to use some of the other types available in PureScript, like `Maybe`.

Suppose we wanted to recreate the `head` function on arrays by using a foreign declaration. In JavaScript, we might write the function as follows:

```
export const head = arr =>
  arr[0];
```

How would we type this function? We might try to give it the type `forall a. Array a -> a`, but for empty arrays, this function returns `undefined`. Therefore, the type `forall a. Array a -> a` does not correctly represent this implementation.

We instead want to return a `Maybe` value to handle this corner case:

```
foreign import maybeHead :: forall a. Array a -> Maybe a
```

But how do we return a `Maybe`? It is tempting to write the following:

```
// Don't do this
import Data_Maybe from '../Data.Maybe'

export const maybeHead = arr => {
  if (arr.length) {
    return Data_Maybe.Just.create(arr[0]);
  } else {
    return Data_Maybe.Nothing.value;
  }
}
```

Importing and using the `Data.Maybe` module directly in the foreign module isn't recommended as it makes our code brittle to changes in the code generator — `create` and `value` are not public APIs. Additionally, doing this can cause problems when using `purs bundle` for dead code elimination.

The recommended approach is to add extra parameters to our FFI-defined function to accept the functions we need.

```
export const maybeHeadImpl = just => nothing => arr => {
  if (arr.length) {
    return just(arr[0]);
  } else {
    return nothing;
  }
};
```

```
foreign import maybeHeadImpl :: forall a. (forall x. x -> Maybe x) -> (forall
x. Maybe x) -> Array a -> Maybe a

maybeHead :: forall a. Array a -> Maybe a
maybeHead arr = maybeHeadImpl Just Nothing arr
```

Note that we wrote:

```
forall a. (forall x. x -> Maybe x) -> (forall x. Maybe x) -> Array a -> Maybe a
```

And not:

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

While both forms work, the latter is more vulnerable to unwanted inputs in place of `Just` and `Nothing`.

For example, in the more vulnerable case, we could call it as follows:

```
maybeHeadImpl (\_ -> Just 1000) (Just 1000) [1,2,3]
```

Which returns `Just 1000` for any array input.

This vulnerability is allowed because `(\_ -> Just 1000)` and `Just 1000` match the signatures of `(a -> Maybe a)` and `Maybe a`, respectively, when `a` is `Int` (based on input array).

In the more secure type signature, even when `a` is determined to be `Int` based on the input array, we still need to provide valid functions matching the signatures involving `forall x`. The *only* option for `(forall x. Maybe x)` is `Nothing`, since a `Just` value would assume a type for `x` and will no longer be valid for all `x`. The only options for `(forall x. x -> Maybe x)` are `Just` (our desired argument) and `(\_ -> Nothing)`, which is the only remaining vulnerability.

Defining Foreign Types

Suppose instead of returning a `Maybe a`, we want to return `arr[0]`. We want a type that represents a value either of type `a` or the `undefined` value (but not `null`). We'll call this type `Undefined a`.

We can define a *foreign type* using a *foreign type declaration*. The syntax is similar to defining a foreign function:

```
foreign import data Undefined :: Type -> Type
```

The `data` keyword here indicates that we are defining a *type*, not a value. Instead of a type

signature, we give the *kind* of the new type. In this case, we declare the kind of `Undefined` to be `Type -> Type`. In other words, `Undefined` is a type constructor.

We can now reuse our original definition for `head`:

```
export const undefinedHead = arr =>
  arr[0];
```

And in the PureScript module:

```
foreign import undefinedHead :: forall a. Array a -> Undefined a
```

The body of the `undefinedHead` function returns `arr[0]`, which may be `undefined`, and the type signature correctly reflects that fact.

This function has the correct runtime representation for its type, but it's quite useless since we have no way to use a value of type `Undefined a`. Well, not exactly. We can use this type in another FFI!

We can write a function that will tell us whether a value is undefined or not:

```
foreign import isUndefined :: forall a. Undefined a -> Boolean
```

This is defined in our foreign JavaScript module as follows:

```
export const isUndefined = value =>
  value === undefined;
```

We can now use `isUndefined` and `undefinedHead` together from PureScript to define a useful function:

```
isEmpty :: forall a. Array a -> Boolean
isEmpty = isUndefined <<< undefinedHead
```

Here, the foreign function we defined is very simple, which means we can benefit from using PureScript's typechecker as much as possible. This is good practice in general: foreign functions should be kept as small as possible, and application logic moved into PureScript code wherever possible.

Exceptions

Another option is to simply throw an exception in the case of an empty array. Strictly speaking, pure functions should not throw exceptions, but we have the flexibility to do so. We indicate the lack of safety in the function name:

```
foreign import unsafeHead :: forall a. Array a -> a
```

In our foreign JavaScript module, we can define `unsafeHead` as follows:

```
export const unsafeHead = arr => {
  if (arr.length) {
    return arr[0];
  } else {
    throw new Error('unsafeHead: empty array');
  }
};
```

Exercises

1. (Medium) Given a record that represents a quadratic polynomial $ax^2 + bx + c = 0$:

   ```
   type Quadratic = {
     a :: Number,
     b :: Number,
     c :: Number
   }
   ```

   Write a JavaScript function `quadraticRootsImpl` and a wrapper `quadraticRoots :: Quadratic -> Pair Complex` that uses the quadratic formula to find the roots of this polynomial. Return the two roots as a `Pair` of `Complex` numbers. *Hint:* Use the `quadraticRoots` wrapper to pass a constructor for `Pair` to `quadraticRootsImpl`.

2. (Medium) Write the function `toMaybe :: forall a. Undefined a -> Maybe a`. This function converts `undefined` to `Nothing` and `a` values to `Just`s.

3. (Difficult) With `toMaybe` in place, we can rewrite `maybeHead` as

   ```
   maybeHead :: forall a. Array a -> Maybe a
   maybeHead = toMaybe <<< undefinedHead
   ```

   Is this a better approach than our previous implementation? *Note:* There is no unit test

for this exercise.

Using Type Class Member Functions

Like our earlier guide on passing the `Maybe` constructor over FFI, this is another case of writing PureScript that calls JavaScript, which calls PureScript functions again. Here we will explore how to pass type class member functions over the FFI.

We start with writing a foreign JavaScript function that expects the appropriate instance of `show` to match the type of `x`.

```
export const boldImpl = show => x =>
  show(x).toUpperCase() + "!!!";
```

Then we write the matching signature:

```
foreign import boldImpl :: forall a. (a -> String) -> a -> String
```

And a wrapper function that passes the correct instance of `show`:

```
bold :: forall a. Show a => a -> String
bold x = boldImpl show x
```

Alternatively, in point-free form:

```
bold :: forall a. Show a => a -> String
bold = boldImpl show
```

We can then call the wrapper:

```
$ spago repl

> import Test.Examples
> import Data.Tuple
> bold (Tuple 1 "Hat")
"(TUPLE 1 \"HAT\")!!!"
```

Here's another example demonstrating passing multiple functions, including a function of multiple arguments (`eq`):

```
export const showEqualityImpl = eq => show => a => b => {
  if (eq(a)(b)) {
    return "Equivalent";
  } else {
    return show(a) + " is not equal to " + show(b);
  }
}
```

```
foreign import showEqualityImpl :: forall a. (a -> a -> Boolean) -> (a ->
String) -> a -> a -> String

showEquality :: forall a. Eq a => Show a => a -> a -> String
showEquality = showEqualityImpl eq show
```

```
$ spago repl

> import Test.Examples
> import Data.Maybe
> showEquality Nothing (Just 5)
"Nothing is not equal to (Just 5)"
```

Effectful Functions

Let's extend our `bold` function to log to the console. Logging is an `Effect`, and `Effect`s are represented in JavaScript as a function of zero arguments, `()` with arrow notation:

```
export const yellImpl = show => x => () =>
  console.log(show(x).toUpperCase() + "!!!");
```

The new foreign import is the same as before, except that the return type changed from `String` to `Effect Unit`.

```
foreign import yellImpl :: forall a. (a -> String) -> a -> Effect Unit

yell :: forall a. Show a => a -> Effect Unit
yell = yellImpl show
```

When testing this in the repl, notice that the string is printed directly to the console (instead of being quoted), and a `unit` value is returned.

```
$ spago repl

> import Test.Examples
> import Data.Tuple
> yell (Tuple 1 "Hat")
(TUPLE 1 "HAT")!!!
unit
```

There are also `EffectFn` wrappers from `Effect.Uncurried`. These are similar to the `Fn` wrappers from `Data.Function.Uncurried` that we've already seen. These wrappers let you call uncurried effectful functions in PureScript.

You'd generally only use these if you want to call existing JavaScript library APIs directly rather than wrapping those APIs in curried functions. So it doesn't make much sense to present an example of uncurried `yell`, where the JavaScript relies on PureScript type class members since you wouldn't find that in the existing JavaScript ecosystem.

Instead, we'll modify our previous `diagonal` example to include logging in addition to returning the result:

```
export const diagonalLog = function(w, h) {
  let result = Math.sqrt(w * w + h * h);
  console.log("Diagonal is " + result);
  return result;
};
```

```
foreign import diagonalLog :: EffectFn2 Number Number Number
```

```
$ spago repl

> import Test.Examples
> import Effect.Uncurried
> runEffectFn2 diagonalLog 3.0 4.0
Diagonal is 5
5.0
```

Asynchronous Functions

Promises in JavaScript translate directly to asynchronous effects in PureScript with the help of the `aff-promise` library. See that library's documentation for more information. We'll just go through a few examples.

Suppose we want to use this JavaScript `wait` promise (or asynchronous function) in our PureScript project. It may be used to delay execution for `ms` milliseconds.

```
const wait = ms => new Promise(resolve => setTimeout(resolve, ms));
```

We just need to export it wrapped as an `Effect` (function of zero arguments):

```
export const sleepImpl = ms => () =>
  wait(ms);
```

Then import it as follows:

```
foreign import sleepImpl :: Int -> Effect (Promise Unit)

sleep :: Int -> Aff Unit
sleep = sleepImpl >>> toAffE
```

We can then run this `Promise` in an `Aff` block like so:

```
$ spago repl

> import Prelude
> import Test.Examples
> import Effect.Class.Console
> import Effect.Aff
> :pa
… launchAff_ do
…   log "waiting"
…   sleep 300
…   log "done waiting"
…
waiting
unit
done waiting
```

Note that asynchronous logging in the repl waits to print until the entire block has finished executing. This code behaves more predictably when run with `spago test` where there is a slight delay *between* prints.

Let's look at another example where we return a value from a promise. This function is written with `async` and `await`, which is just syntactic sugar for promises.

```
async function diagonalWait(delay, w, h) {
  await wait(delay);
  return Math.sqrt(w * w + h * h);
}

export const diagonalAsyncImpl = delay => w => h => () =>
  diagonalWait(delay, w, h);
```

Since we're returning a `Number`, we represent this type in the `Promise` and `Aff` wrappers:

```
foreign import diagonalAsyncImpl :: Int -> Number -> Number -> Effect (Promise
Number)

diagonalAsync :: Int -> Number -> Number -> Aff Number
diagonalAsync i x y = toAffE $ diagonalAsyncImpl i x y
```

```
$ spago repl

import Prelude
import Test.Examples
import Effect.Class.Console
import Effect.Aff
> :pa
… launchAff_ do
…   res <- diagonalAsync 300 3.0 4.0
…   logShow res
…
unit
5.0
```

Exercises

Exercises for the above sections are still on the ToDo list. If you have any ideas for good exercises, please make a suggestion.

JSON

There are many reasons to use JSON in an application; for example, it's a common means of communicating with web APIs. This section will discuss other use-cases, too, beginning with a technique to improve type safety when passing structural data over the FFI.

Let's revisit our earlier FFI functions `cumulativeSums` and `addComplex` and introduce a bug to each:

```
export const cumulativeSumsBroken = arr => {
  let sum = 0
  let sums = []
  arr.forEach(x => {
    sum += x;
    sums.push(sum);
  });
  sums.push("Broken"); // Bug
  return sums;
};

export const addComplexBroken = a => b => {
  return {
    real: a.real + b.real,
    broken: a.imag + b.imag // Bug
  }
};
```

We can use the original type signatures, and the code will still compile, despite the incorrect return types.

```
foreign import cumulativeSumsBroken :: Array Int -> Array Int

foreign import addComplexBroken :: Complex -> Complex -> Complex
```

We can even execute the code, which might either produce unexpected results or a runtime error:

```
$ spago repl

> import Test.Examples
> import Data.Foldable (sum)

> sums = cumulativeSumsBroken [1, 2, 3]
> sums
[1,3,6,Broken]
> sum sums
0

> complex = addComplexBroken { real: 1.0, imag: 2.0 } { real: 3.0, imag: 4.0 }
> complex.real
4.0
> complex.imag + 1.0
NaN
> complex.imag
  var str = n.toString();
              ^
TypeError: Cannot read property 'toString' of undefined
```

For example, our resulting `sums` is no-longer a valid `Array Int`, now that a `String` is included in the Array. And further operations produce unexpected behavior, rather than an outright error, as the `sum` of these `sums` is `0` rather than `10`. This could be a difficult bug to track down!

Likewise, there are no errors when calling `addComplexBroken`; however, accessing the `imag` field of our `Complex` result will either produce unexpected behavior (returning `NaN` instead of `7.0`), or a non-obvious runtime error.

Let's use JSON to make our PureScript code more impervious to bugs in JavaScript code.

The `argonaut` library contains the JSON decoding and encoding capabilities we need. That library has excellent documentation, so we will only cover basic usage in this book.

If we create an alternate foreign import that defines the return type as `Json`:

```
foreign import cumulativeSumsJson :: Array Int -> Json
foreign import addComplexJson :: Complex -> Complex -> Json
```

Note that we're simply pointing to our existing broken functions:

```
export const cumulativeSumsJson = cumulativeSumsBroken
export const addComplexJson = addComplexBroken
```

And then write a wrapper to decode the returned foreign `Json` value:

```
cumulativeSumsDecoded :: Array Int -> Either JsonDecodeError (Array Int)
cumulativeSumsDecoded arr = decodeJson $ cumulativeSumsJson arr

addComplexDecoded :: Complex -> Complex -> Either JsonDecodeError Complex
addComplexDecoded a b = decodeJson $ addComplexJson a b
```

Then any values that can't be successfully decoded to our return type appear as a `Left` error `String`:

```
$ spago repl

> import Test.Examples

> cumulativeSumsDecoded [1, 2, 3]
(Left "Couldn't decode Array (Failed at index 3): Value is not a Number")

> addComplexDecoded { real: 1.0, imag: 2.0 } { real: 3.0, imag: 4.0 }
(Left "JSON was missing expected field: imag")
```

If we call the working versions, a `Right` value is returned.

Try this yourself by modifying `test/Examples.js` with the following change to point to the working versions before running the next repl block.

```
export const cumulativeSumsJson = cumulativeSums
export const addComplexJson = addComplex
```

```
$ spago repl

> import Test.Examples

> cumulativeSumsDecoded [1, 2, 3]
(Right [1,3,6])

> addComplexDecoded { real: 1.0, imag: 2.0 } { real: 3.0, imag: 4.0 }
(Right { imag: 6.0, real: 4.0 })
```

Using JSON is also the easiest way to pass other structural types, such as `Map` and `Set`, through the FFI. Since JSON only consists of booleans, numbers, strings, arrays, and objects of other JSON values, we can't write a `Map` and `Set` directly in JSON. But we can represent these structures as arrays (assuming the keys and values can also be represented in JSON) and then decode them back to `Map` or `Set`.

Here's an example of a foreign function signature that modifies a `Map` of `String` keys and `Int` values, along with the wrapper function that handles JSON encoding and decoding.

```
foreign import mapSetFooJson :: Json -> Json

mapSetFoo :: Map String Int -> Either JsonDecodeError (Map String Int)
mapSetFoo json = decodeJson $ mapSetFooJson $ encodeJson json
```

Note that this is a prime use case for function composition. Both of these alternatives are equivalent to the above:

```
mapSetFoo :: Map String Int -> Either JsonDecodeError (Map String Int)
mapSetFoo = decodeJson <<< mapSetFooJson <<< encodeJson

mapSetFoo :: Map String Int -> Either JsonDecodeError (Map String Int)
mapSetFoo = encodeJson >>> mapSetFooJson >>> decodeJson
```

Here is the JavaScript implementation. Note the `Array.from` step, which is necessary to convert the JavaScript `Map` into a JSON-friendly format before decoding converts it back to a PureScript `Map`.

```
export const mapSetFooJson = j => {
  let m = new Map(j);
  m.set("Foo", 42);
  return Array.from(m);
};
```

Now we can send and receive a `Map` over the FFI:

```
$ spago repl

> import Test.Examples
> import Data.Map
> import Data.Tuple

> myMap = fromFoldable [ Tuple "hat" 1, Tuple "cat" 2 ]

> :type myMap
Map String Int

> myMap
(fromFoldable [(Tuple "cat" 2),(Tuple "hat" 1)])

> mapSetFoo myMap
(Right (fromFoldable [(Tuple "Foo" 42),(Tuple "cat" 2),(Tuple "hat" 1)]))
```

Exercises

1. (Medium) Write a JavaScript function and PureScript wrapper `valuesOfMap :: Map String Int -> Either JsonDecodeError (Set Int)` that returns a `Set` of all the values in a `Map`. *Hint*: The `.values()` instance method for Map may be useful in your JavaScript code.

2. (Easy) Write a new wrapper for the previous JavaScript function with the signature `valuesOfMapGeneric :: forall k v. Map k v -> Either JsonDecodeError (Set v)` so it works with a wider variety of maps. Note that you'll need to add some type class constraints for `k` and `v`. The compiler will guide you.

3. (Medium) Rewrite the earlier `quadraticRoots` function as `quadraticRootsSet` that returns the `Complex` roots as a `Set` via JSON (instead of as a `Pair`).

4. (Difficult) Rewrite the earlier `quadraticRoots` function as `quadraticRootsSafe` that uses JSON to pass the `Pair` of `Complex` roots over FFI. Don't use the `Pair` constructor in JavaScript, but instead, just return the pair in a decoder-compatible format. *Hint*: You'll need to write a `DecodeJson` instance for `Pair`. Consult the argonaut docs for instruction on writing your own decode instance. Their decodeJsonTuple instance may also be a helpful reference. Note that you'll need a `newtype` wrapper for `Pair` to avoid creating an "orphan instance".

5. (Medium) Write a `parseAndDecodeArray2D :: String -> Either String (Array (Array Int))` function to parse and decode a JSON string containing a 2D array, such as `"[[1, 2, 3], [4, 5], [6]]"`. *Hint*: You'll need to use `jsonParser` to convert the `String` into `Json` before decoding.

6. (Medium) The following data type represents a binary tree with values at the leaves:

   ```
   data Tree a
     = Leaf a
     | Branch (Tree a) (Tree a)
   ```

   Derive generic `EncodeJson` and `DecodeJson` instances for the `Tree` type. Consult the argonaut docs for instructions on how to do this. Note that you'll also need generic instances of `Show` and `Eq` to enable unit testing for this exercise, but those should be straightforward to implement after tackling the JSON instances.

7. (Difficult) The following `data` type should be represented directly in JSON as either an integer or a string:

```
data IntOrString
  = IntOrString_Int Int
  | IntOrString_String String
```

Write instances of `EncodeJson` and `DecodeJson` for the `IntOrString` data type which implement this behavior. *Hint*: The `alt` operator from `Control.Alt` may be helpful.

Address book

In this section, we will apply our newly-acquired FFI and JSON knowledge to build on our address book example from Chapter 8. We will add the following features:

- A Save button at the bottom of the form that, when clicked, serializes the state of the form to JSON and saves it in local storage.
- Automatic retrieval of the JSON document from local storage upon page reload. The form fields are populated with the contents of this document.
- A pop-up alert if there is an issue saving or loading the form state.

We'll start by creating FFI wrappers for the following Web Storage APIs in our `Effect.Storage` module:

- `setItem` takes a key and a value (both strings), and returns a computation which stores (or updates) the value in local storage at the specified key.
- `getItem` takes a key, and attempts to retrieve the associated value from local storage. However, since the `getItem` method on `window.localStorage` can return `null`, the return type is not `String`, but `Json`.

```
foreign import setItem :: String -> String -> Effect Unit

foreign import getItem :: String -> Effect Json
```

Here is the corresponding JavaScript implementation of these functions in `Effect/Storage.js`:

```
export const setItem = key => value => () =>
  window.localStorage.setItem(key, value);

export const getItem = key => () =>
  window.localStorage.getItem(key);
```

We'll create a save button like so:

```
saveButton :: R.JSX
saveButton =
  D.label
    { className: "form-group row col-form-label"
    , children:
        [ D.button
            { className: "btn-primary btn"
            , onClick: handler_ validateAndSave
            , children: [ D.text "Save" ]
            }
        ]
    }
```

And write our validated `person` as a JSON string with `setItem` in the `validateAndSave` function:

```
validateAndSave :: Effect Unit
validateAndSave = do
  log "Running validators"
  case validatePerson' person of
    Left errs -> log $ "There are " <> show (length errs) <> " validation
errors."
    Right validPerson -> do
      setItem "person" $ stringify $ encodeJson validPerson
      log "Saved"
```

Note that if we attempt to compile at this stage, we'll encounter the following error:

```
  No type class instance was found for
    Data.Argonaut.Encode.Class.EncodeJson PhoneType
```

This is because `PhoneType` in the `Person` record needs an `EncodeJson` instance. We'll also derive a generic encode instance and a decode instance while we're at it. More information on how this works is available in the argonaut docs:

```
import Data.Argonaut (class DecodeJson, class EncodeJson)
import Data.Argonaut.Decode.Generic (genericDecodeJson)
import Data.Argonaut.Encode.Generic (genericEncodeJson)
import Data.Generic.Rep (class Generic)

derive instance Generic PhoneType _

instance EncodeJson PhoneType where encodeJson = genericEncodeJson
instance DecodeJson PhoneType where decodeJson = genericDecodeJson
```

Now we can save our `person` to local storage, but this isn't very useful unless we can retrieve the data. We'll tackle that next.

We'll start with retrieving the "person" string from local storage:

```
item <- getItem "person"
```

Then we'll create a helper function to convert the string from local storage to our `Person` record. Note that this string in storage may be `null`, so we represent it as a foreign `Json` until it is successfully decoded as a `String`. There are a number of other conversion steps along the way – each of which returns an `Either` value, so it makes sense to organize these together in a `do` block.

```
processItem :: Json -> Either String Person
processItem item = do
  jsonString <- decodeJson item
  j          <- jsonParser jsonString
  decodeJson j
```

Then we inspect this result to see if it succeeded. If it fails, we'll log the errors and use our default `examplePerson`, otherwise, we'll use the person retrieved from local storage.

```
initialPerson <- case processItem item of
  Left  err -> do
    log $ "Error: " <> err <> ". Loading examplePerson"
    pure examplePerson
  Right p   -> pure p
```

Finally, we'll pass this `initialPerson` to our component via the `props` record:

```
-- Create JSX node from react component.
app = element addressBookApp { initialPerson }
```

And pick it up on the other side to use in our state hook:

```
mkAddressBookApp :: Effect (ReactComponent { initialPerson :: Person })
mkAddressBookApp =
  reactComponent "AddressBookApp" \props -> R.do
    Tuple person setPerson <- useState props.initialPerson
```

As a finishing touch, we'll improve the quality of our error messages by appending to the `String` of each `Left` value with `lmap`.

```
processItem :: Json -> Either String Person
processItem item = do
  jsonString <- lmap ("No string in local storage: " <> _) $ decodeJson item
  j          <- lmap ("Cannot parse JSON string: "   <> _) $ jsonParser
jsonString
  lmap               ("Cannot decode Person: "       <> _) $ decodeJson j
```

Only the first error should ever occur during the normal operation of this app. You can trigger the other errors by opening your web browser's dev tools, editing the saved "person" string in local storage, and refreshing the page. How you modify the JSON string determines which error is triggered. See if you can trigger each of them.

That covers local storage. Next, we'll implement the `alert` action, similar to the `log` action from the `Effect.Console` module. The only difference is that the `alert` action uses the `window.alert` method, whereas the `log` action uses the `console.log` method. As such, `alert` can only be used in environments where `window.alert` is defined, such as a web browser.

```
foreign import alert :: String -> Effect Unit
```

```
export const alert = msg => () =>
  window.alert(msg);
```

We want this alert to appear when either:

- A user attempts to save a form with validation errors.
- The state cannot be retrieved from local storage.

That is accomplished by simply replacing `log` with `alert` on these lines:

```
Left errs -> alert $ "There are " <> show (length errs) <> " validation
errors."
```

```
alert $ "Error: " <> err <> ". Loading examplePerson"
```

Exercises

1. (Easy) Write a wrapper for the `removeItem` method on the `localStorage` object, and add your foreign function to the `Effect.Storage` module.
2. (Medium) Add a "Reset" button that, when clicked, calls the newly-created `removeItem` function to delete the "person" entry from local storage.

3. (Easy) Write a wrapper for the `confirm` method on the JavaScript `Window` object, and add your foreign function to the `Effect.Alert` module.
4. (Medium) Call this `confirm` function when a users clicks the "Reset" button to ask if they're sure they want to reset their address book.

Conclusion

In this chapter, we've learned how to work with foreign JavaScript code from PureScript, and we've seen the issues involved with writing trustworthy code using the FFI:

- We've seen the importance of ensuring that foreign functions have correct representations.
- We learned how to deal with corner cases like null values and other types of JavaScript data by using foreign types or the `Json` data type.
- We saw how to safely serialize and deserialize JSON data.

For more examples, the `purescript`, `purescript-contrib`, and `purescript-node` GitHub organizations provide plenty of examples of libraries that use the FFI. In the remaining chapters, we will see some of these libraries put to use to solve real-world problems in a type-safe way.

Addendum

Calling PureScript from JavaScript

Calling a PureScript function from JavaScript is very simple, at least for functions with simple types.

Let's take the following simple module as an example:

```
module Test where

gcd :: Int -> Int -> Int
gcd 0 m = m
gcd n 0 = n
gcd n m
  | n > m     = gcd (n - m) m
  | otherwise = gcd (m - n) n
```

This function finds the greatest common divisor of two numbers by repeated subtraction. It is a nice example of a case where you might like to use PureScript to define the function, but have a requirement to call it from JavaScript: it is simple to define this function in PureScript using pattern matching and recursion, and the implementor can benefit from the use of the type checker.

To understand how this function can be called from JavaScript, it is important to realize that PureScript functions always get turned into JavaScript functions of a single argument, so we need to apply its arguments one-by-one:

```
import Test from 'Test.js';
Test.gcd(15)(20);
```

Here, I assume the code was compiled with `spago build`, which compiles PureScript modules to ES modules. For that reason, I could reference the `gcd` function on the `Test` object, after importing the `Test` module using `import`.

You can also use the `spago bundle-app` and `spago bundle-module` commands to bundle your generated JavaScript into a single file. Consult the documentation for more information.

Understanding Name Generation

PureScript aims to preserve names during code generation as much as possible. In particular, most identifiers that are neither PureScript nor JavaScript keywords can be expected to be preserved, at least for names of top-level declarations.

If you decide to use a JavaScript keyword as an identifier, the name will be escaped with a double dollar symbol. For example,

```
null = []
```

Generates the following JavaScript:

```
var $$null = [];
```

In addition, if you would like to use special characters in your identifier names, they will be escaped using a single dollar symbol. For example,

```
example' = 100
```

Generates the following JavaScript:

```
var example$prime = 100;
```

Where compiled PureScript code is intended to be called from JavaScript, it is recommended that identifiers only use alphanumeric characters and avoid JavaScript keywords. If user-defined operators are provided for use in PureScript code, it is good practice to provide an alternative function with an alphanumeric name for use in JavaScript.

Runtime Data Representation

Types allow us to reason at compile-time that our programs are "correct" in some sense – that is, they will not break at runtime. But what does that mean? In PureScript, it means that the type of an expression should be compatible with its representation at runtime.

For that reason, it is important to understand the representation of data at runtime to be able to use PureScript and JavaScript code together effectively. This means that for any given PureScript expression, we should be able to understand the behavior of the value it will evaluate to at runtime.

The good news is that PureScript expressions have particularly simple representations at runtime. It should always be possible to understand the runtime data representation of an expression by considering its type.

For simple types, the correspondence is almost trivial. For example, if an expression has the type `Boolean`, then its value `v` at runtime should satisfy `typeof v === 'boolean'`. That is, expressions of type `Boolean` evaluate to one of the (JavaScript) values `true` or `false`. In particular, there is no PureScript expression of type `Boolean` which evaluates to `null` or `undefined`.

A similar law holds for expressions of type `Int`, `Number`, and `String` – expressions of type `Int` or `Number` evaluate to non-null JavaScript numbers, and expressions of type `String` evaluate to non-null JavaScript strings. Expressions of type `Int` will evaluate to integers at runtime, even though they cannot be distinguished from values of type `Number` by using `typeof`.

What about `Unit`? Well, since `Unit` has only one inhabitant (`unit`) and its value is not observable, it doesn't matter what it's represented with at runtime. Old code tends to represent it using `{}`. Newer code, however, tends to use `undefined`. So, although it doesn't matter what you use to represent `Unit`, it is recommended to use `undefined` (not returning anything from a function also returns `undefined`).

What about some more complex types?

As we have already seen, PureScript functions correspond to JavaScript functions of a single argument. More precisely, if an expression `f` has type `a -> b` for some types `a` and `b`, and an expression `x` evaluates to a value with the correct runtime representation for type `a`, then `f` evaluates to a JavaScript function, which, when applied to the result of evaluating `x`, has the correct runtime representation for type `b`. As a simple example, an expression of type `String -> String` evaluates to a function that takes non-null JavaScript strings to non-null JavaScript strings.

As you might expect, PureScript's arrays correspond to JavaScript arrays. But remember – PureScript arrays are homogeneous, so every element has the same type. Concretely, if a PureScript expression `e` has type `Array a` for some type `a`, then `e` evaluates to a (non-null) JavaScript array, all of whose elements have the correct runtime representation for type `a`.

We've already seen that PureScript's records evaluate to JavaScript objects. As for functions and arrays, we can reason about the runtime representation of data in a record's fields by considering the types associated with its labels. Of course, the fields of a record are not required to be of the same type.

Representing ADTs

For every constructor of an algebraic data type, the PureScript compiler creates a new JavaScript object type by defining a function. Its constructors correspond to functions that create new JavaScript objects based on those prototypes.

For example, consider the following simple ADT:

```
data ZeroOrOne a = Zero | One a
```

The PureScript compiler generates the following code:

```
function One(value0) {
    this.value0 = value0;
};

One.create = function (value0) {
    return new One(value0);
};

function Zero() {
};

Zero.value = new Zero();
```

Here, we see two JavaScript object types: `Zero` and `One`. It is possible to create values of each type by using JavaScript's `new` keyword. For constructors with arguments, the compiler stores the associated data in fields called `value0`, `value1`, etc.

The PureScript compiler also generates helper functions. For constructors with no arguments, the compiler generates a `value` property, which can be reused instead of using the `new` operator repeatedly. For constructors with one or more arguments, the compiler generates a `create` function, which takes arguments with the appropriate representation and applies the appropriate constructor.

What about constructors with more than one argument? In that case, the PureScript compiler also creates a new object type, and a helper function. This time, however, the helper function is a curried function of two arguments. For example, this algebraic data type:

```
data Two a b = Two a b
```

Generates this JavaScript code:

```
function Two(value0, value1) {
    this.value0 = value0;
    this.value1 = value1;
};

Two.create = function (value0) {
    return function (value1) {
        return new Two(value0, value1);
    };
};
```

Here, values of the object type `Two` can be created using the `new` keyword or by using the `Two.create` function.

The case of newtypes is slightly different. Recall that a newtype is like an algebraic data type,

restricted to having a single constructor taking a single argument. In this case, the runtime representation of the newtype is the same as its argument type.

For example, this newtype represents telephone numbers is represented as a JavaScript string at runtime:

```
newtype PhoneNumber = PhoneNumber String
```

This is useful for designing libraries since newtypes provide an additional layer of type safety without the runtime overhead of another function call.

Representing Quantified Types

Expressions with quantified (polymorphic) types have restrictive representations at runtime. In practice, there are relatively few expressions with a given quantified type, but we can reason about them quite effectively.

Consider this polymorphic type, for example:

```
forall a. a -> a
```

What sort of functions have this type? Well, there is certainly one function with this type:

```
identity :: forall a. a -> a
identity a = a
```

> Note that the actual `identity` function defined in `Prelude` has a slightly different type.

In fact, the `identity` function is the *only* (total) function with this type! This certainly seems to be the case (try writing an expression with this type that is not observably equivalent to `identity`), but how can we be sure? We can be sure by considering the runtime representation of the type.

What is the runtime representation of a quantified type `forall a. t`? Well, any expression with the runtime representation for this type must have the correct runtime representation for the type `t` for any choice of type `a`. In our example above, a function of type `forall a. a -> a` must have the correct runtime representation for the types `String -> String`, `Number -> Number`, `Array Boolean -> Array Boolean`, and so on. It must take strings to

strings, numbers to numbers, etc.

But that is not enough – the runtime representation of a quantified type is more strict than this. We require any expression to be *parametrically polymorphic* – that is, it cannot use any information about the type of its argument in its implementation. This additional condition prevents problematic implementations such as the following JavaScript function from inhabiting a polymorphic type:

```
function invalid(a) {
    if (typeof a === 'string') {
        return "Argument was a string.";
    } else {
        return a;
    }
}
```

Certainly, this function takes strings to strings, numbers to numbers, etc. But it does not meet the additional condition, since it inspects the (runtime) type of its argument, so this function would not be a valid inhabitant of the type `forall a. a -> a`.

Without being able to inspect the runtime type of our function argument, our only option is to return the argument unchanged. So `identity` is indeed the only inhabitant of the type `forall a. a -> a`.

A full discussion of *parametric polymorphism* and *parametricity* is beyond the scope of this book. Note, however, that since PureScript's types are *erased* at runtime, a polymorphic function in PureScript *cannot* inspect the runtime representation of its arguments (without using the FFI), so this representation of polymorphic data is appropriate.

Representing Constrained Types

Functions with a type class constraint have an interesting representation at runtime. Because the function's behavior might depend on the type class instance chosen by the compiler, the function is given an additional argument, called a *type class dictionary*, which contains the implementation of the type class functions provided by the chosen instance.

For example, here is a simple PureScript function with a constrained type that uses the `Show` type class:

```
shout :: forall a. Show a => a -> String
shout a = show a <> "!!!"
```

The generated JavaScript looks like this:

```
var shout = function (dict) {
    return function (a) {
        return show(dict)(a) + "!!!";
    };
};
```

Notice that `shout` is compiled to a (curried) function of two arguments, not one. The first argument `dict` is the type class dictionary for the `Show` constraint. `dict` contains the implementation of the `show` function for the type `a`.

We can call this function from JavaScript by passing an explicit type class dictionary from `Data.Show` as the first parameter:

```
import { showNumber } from 'Data.Show'

shout(showNumber)(42);
```

Exercises

1. (Easy) What are the runtime representations of these types?

   ```
   forall a. a
   forall a. a -> a -> a
   forall a. Ord a => Array a -> Boolean
   ```

   What can you say about the expressions which have these types?

2. (Medium) Try using the functions defined in the `arrays` package, calling them from JavaScript, by compiling the library using `spago build` and importing modules using the `import` function in NodeJS. *Hint*: you may need to configure the output path so that the generated ES modules are available on the NodeJS module path.

Representing Side Effects

The `Effect` monad is also defined as a foreign type. Its runtime representation is quite simple – an expression of type `Effect a` should evaluate to a JavaScript function of **no arguments**, which performs any side-effects and returns a value with the correct runtime representation for type `a`.

The definition of the `Effect` type constructor is given in the `Effect` module as follows:

```
foreign import data Effect :: Type -> Type
```

As a simple example, consider the `random` function defined in the `random` package. Recall that its type was:

```
foreign import random :: Effect Number
```

The definition of the `random` function is given here:

```
export const random = Math.random;
```

Notice that the `random` function is represented at runtime as a function of no arguments. It performs the side effect of generating a random number, returns it, and the return value matches the runtime representation of the `Number` type: it is a non-null JavaScript number.

As a slightly more interesting example, consider the `log` function defined by the `Effect.Console` module in the `console` package. The `log` function has the following type:

```
foreign import log :: String -> Effect Unit
```

And here is its definition:

```
export const log = function (s) {
  return function () {
    console.log(s);
  };
};
```

The representation of `log` at runtime is a JavaScript function of a single argument, returning a function of no arguments. The inner function performs the side-effect of writing a message to the console.

Expressions of type `Effect a` can be invoked from JavaScript like regular JavaScript methods. For example, since the `main` function is required to have type `Effect a` for some type `a`, it can be invoked as follows:

```
import { main } from 'Main'

main();
```

When using `spago bundle-app --to` or `spago run`, this call to `main` is generated automatically whenever the `Main` module is defined.

Monadic Adventures

Chapter Goals

The goal of this chapter will be to learn about *monad transformers*, which provide a way to combine side-effects provided by different monads. The motivating example will be a text adventure game that can be played on the console in NodeJS. The various side-effects of the game (logging, state, and configuration) will all be provided by a monad transformer stack.

Project Setup

This module's project introduces the following new dependencies:

- `ordered-collections`, which provides data types for immutable maps and sets
- `transformers`, which provides implementations of standard monad transformers
- `node-readline`, which provides FFI bindings to the `readline` interface provided by NodeJS
- `optparse`, which provides applicative parsers for processing command-line arguments

How To Play The Game

To run the project, use `spago run`

By default, you will see a usage message:

```
Monadic Adventures! A game to learn monad transformers

Usage: run.js (-p|--player <player name>) [-d|--debug]
  Play the game as <player name>

Available options:
  -p,--player <player name>
                           The player's name <String>
  -d,--debug               Use debug mode
  -h,--help                Show this help text
```

To provide command line arguments, you can either call `spago run` with the `-a` option to

pass additional arguments directly to your application or call `spago bundle-app`, which will create an index.js file that can be run directly with `node`.

For example, to provide the player name using the `-p` option:

```
$ spago run -a "-p Phil"
>


$ spago bundle-app
$ node index.js -p Phil
>
```

From the prompt, you can enter commands like `look`, `inventory`, `take`, `use`, `north`, `south`, `east`, and `west`. There is also a `debug` command, which can print the game state when the `--debug` command line option is provided.

The game is played on a two-dimensional grid, and the player moves by issuing commands `north`, `south`, `east`, and `west`. The game contains a collection of items that can either be in the player's possession (in the user's *inventory*) or on the game grid at some location. Items can be picked up by the player using the `take` command.

For reference, here is a complete walkthrough of the game:

```
$ spago run -a "-p Phil"

> look
You are at (0, 0)
You are in a dark forest. You see a path to the north.
You can see the Matches.

> take Matches
You now have the Matches

> north
> look
You are at (0, 1)
You are in a clearing.
You can see the Candle.

> take Candle
You now have the Candle

> inventory
You have the Candle.
You have the Matches.

> use Matches
You light the candle.
Congratulations, Phil!
You win!
```

The game is very simple, but the aim of the chapter is to use the `transformers` package to build a library that will enable rapid development of this type of game.

The State Monad

We will start by looking at some of the monads provided by the `transformers` package.

The first example is the `State` monad, which provides a way to model *mutable state* in pure code. We have already seen an approach to a mutable state provided by the `Effect` monad. `State` provides an alternative.

The `State` type constructor takes two type parameters: the type `s` of the state and the return type `a`. Even though we speak of the "`State` monad", the instance of the `Monad` type class is actually provided for the `State s` type constructor for any type `s`.

The `Control.Monad.State` module provides the following API:

```
get     :: forall s.             State s s
gets    :: forall s. (s -> a) -> State s a
put     :: forall s. s        -> State s Unit
modify  :: forall s. (s -> s) -> State s s
modify_ :: forall s. (s -> s) -> State s Unit
```

Note that these API signatures are presented in a simplified form using the `State` type constructor for now. The actual API involves `MonadState`, which we'll cover in the later "Type Classes" section of this chapter, so don't worry if you see different signatures in your IDE tooltips or on Pursuit.

Let's see an example. One use of the `State` monad might be to add the values in an array of integers to the current state. We could do that by choosing `Int` as the state type `s` and using `traverse_` to traverse the array, with a call to `modify` for each array element:

```
import Data.Foldable (traverse_)
import Control.Monad.State
import Control.Monad.State.Class

sumArray :: Array Int -> State Int Unit
sumArray = traverse_ \n -> modify \sum -> sum + n
```

The `Control.Monad.State` module provides three functions for running a computation in the `State` monad:

```
evalState :: forall s a. State s a -> s -> a
execState :: forall s a. State s a -> s -> s
runState  :: forall s a. State s a -> s -> Tuple a s
```

Each function takes an initial state of type `s` and a computation of type `State s a`. `evalState` only returns the return value, `execState` only returns the final state, and `runState` returns both, expressed as a value of type `Tuple a s`.

Given the `sumArray` function above, we could use `execState` in PSCi to sum the numbers in several arrays as follows:

```
> :paste
… execState (do
…   sumArray [1, 2, 3]
…   sumArray [4, 5]
…   sumArray [6]) 0
… ^D
21
```

Exercises

1. (Easy) What is the result of replacing `execState` with `runState` or `evalState` in our example above?

2. (Medium) A string of parentheses is *balanced* if it is obtained by either concatenating zero-or-more shorter balanced strings or wrapping a shorter balanced string in a pair of parentheses.

   Use the `State` monad and the `traverse_` function to write a function

   ```
   testParens :: String -> Boolean
   ```

   which tests whether or not a `String` of parentheses is balanced by keeping track of the number of opening parentheses that have not been closed. Your function should work as follows:

   ```
   > testParens ""
   true

   > testParens "((()(()))"
   true

   > testParens ")"
   false

   > testParens "(()("
   false
   ```

   Hint: you may like to use the `toCharArray` function from the `Data.String.CodeUnits` module to turn the input string into an array of characters.

The Reader Monad

Another monad provided by the `transformers` package is the `Reader` monad. This monad provides the ability to read from a global configuration. Whereas the `State` monad provides the ability to read and write a single piece of mutable state, the `Reader` monad only provides the ability to read a single piece of data.

The `Reader` type constructor takes two type arguments: a type `r` which represents the configuration type, and the return type `a`.

The `Control.Monad.Reader` module provides the following API:

```
ask   :: forall r. Reader r r
local :: forall r a. (r -> r) -> Reader r a -> Reader r a
```

The `ask` action can be used to read the current configuration, and the `local` action can be used to run a computation with a modified configuration.

For example, suppose we were developing an application controlled by permissions, and we wanted to use the `Reader` monad to hold the current user's permissions object. We might choose the type `r` to be some type `Permissions` with the following API:

```
hasPermission :: String -> Permissions -> Boolean
addPermission :: String -> Permissions -> Permissions
```

Whenever we wanted to check if the user had a particular permission, we could use `ask` to retrieve the current permissions object. For example, only administrators might be allowed to create new users:

```
createUser :: Reader Permissions (Maybe User)
createUser = do
  permissions <- ask
  if hasPermission "admin" permissions
    then map Just newUser
    else pure Nothing
```

To elevate the user's permissions, we might use the `local` action to modify the `Permissions` object during the execution of some computation:

```
runAsAdmin :: forall a. Reader Permissions a -> Reader Permissions a
runAsAdmin = local (addPermission "admin")
```

Then we could write a function to create a new user, even if the user did not have the `admin` permission:

```
createUserAsAdmin :: Reader Permissions (Maybe User)
createUserAsAdmin = runAsAdmin createUser
```

To run a computation in the `Reader` monad, the `runReader` function can be used to provide the global configuration:

```
runReader :: forall r a. Reader r a -> r -> a
```

Exercises

In these exercises, we will use the `Reader` monad to build a small library for rendering documents with indentation. The "global configuration" will be a number indicating the current indentation level:

```
type Level = Int

type Doc = Reader Level String
```

1. (Easy) Write a function `line` that renders a function at the current indentation level. Your function should have the following type:

   ```
   line :: String -> Doc
   ```

   Hint: use the `ask` function to read the current indentation level. The `power` function from `Data.Monoid` may be helpful too.

2. (Easy) Use the `local` function to write a function

   ```
   indent :: Doc -> Doc
   ```

   which increases the indentation level for a block of code.

3. (Medium) Use the `sequence` function defined in `Data.Traversable` to write a function

   ```
   cat :: Array Doc -> Doc
   ```

   which concatenates a collection of documents, separating them with new lines.

4. (Medium) Use the `runReader` function to write a function

   ```
   render :: Doc -> String
   ```

   which renders a document as a String.

You should now be able to use your library to write simple documents as follows:

```
render $ cat
  [ line "Here is some indented text:"
  , indent $ cat
      [ line "I am indented"
      , line "So am I"
      , indent $ line "I am even more indented"
      ]
  ]
```

The Writer Monad

The `Writer` monad allows accumulating a secondary value in addition to the return value of a computation.

A common use case is to accumulate a log of type `String` or `Array String`, but the `Writer` monad is more general than this. It can accumulate a value in any monoid, so it might be used to keep track of an integer total using the `Additive Int` monoid or to track whether any of several intermediate `Boolean` values were true using the `Disj Boolean` monoid.

The `Writer` type constructor takes two type arguments: a type `w` that should be an instance of the `Monoid` type class, and the return type `a`.

The key element of the `Writer` API is the `tell` function:

```
tell :: forall w a. Monoid w => w -> Writer w Unit
```

The `tell` action appends the provided value to the current accumulated result.

As an example, let's add a log to an existing function using the `Array String` monoid. Consider our previous implementation of the *greatest common divisor* function:

```
gcd :: Int -> Int -> Int
gcd n 0 = n
gcd 0 m = m
gcd n m = if n > m
            then gcd (n - m) m
            else gcd n (m - n)
```

We could add a log to this function by changing the return type to `Writer (Array String) Int`:

```
import Control.Monad.Writer
import Control.Monad.Writer.Class

gcdLog :: Int -> Int -> Writer (Array String) Int
```

We only have to change our function slightly to log the two inputs at each step:

```
gcdLog n 0 = pure n
gcdLog 0 m = pure m
gcdLog n m = do
  tell ["gcdLog " <> show n <> " " <> show m]
  if n > m
    then gcdLog (n - m) m
    else gcdLog n (m - n)
```

We can run a computation in the `Writer` monad by using either of the `execWriter` or `runWriter` functions:

```
execWriter :: forall w a. Writer w a -> w
runWriter  :: forall w a. Writer w a -> Tuple a w
```

Just like in the case of the `State` monad, `execWriter` only returns the accumulated log, whereas `runWriter` returns both the log and the result.

We can test our modified function in PSCi:

```
> import Control.Monad.Writer
> import Control.Monad.Writer.Class

> runWriter (gcdLog 21 15)
Tuple 3 ["gcdLog 21 15","gcdLog 6 15","gcdLog 6 9","gcdLog 6 3","gcdLog 3 3"]
```

Exercises

1. (Medium) Rewrite the `sumArray` function above using the `Writer` monad and the `Additive Int` monoid from the `monoid` package.

2. (Medium) The *Collatz* function is defined on natural numbers n as $n/2$ when n is even and $3n + 1$ when n is odd. For example, the iterated Collatz sequence starting at 10 is as follows:

```
10, 5, 16, 8, 4, 2, 1, ...
```

It is conjectured that the iterated Collatz sequence always reaches 1 after some finite number of applications of the Collatz function.

Write a function that uses recursion to calculate how many iterations of the Collatz function are required before the sequence reaches 1.

Modify your function to use the `Writer` monad to log each application of the Collatz function.

Monad Transformers

Each of the three monads above: `State`, `Reader`, and `Writer`, are also examples of so-called *monad transformers*. The equivalent monad transformers are called `StateT`, `ReaderT`, and `WriterT`, respectively.

What is a monad transformer? Well, as we have seen, a monad augments PureScript code with some type of side effect, which can be interpreted in PureScript by using the appropriate handler (`runState`, `runReader`, `runWriter`, etc.) This is fine if we only need to use *one* side-effect. However, it is often useful to use more than one side-effect at once. For example, we might want to use `Reader` together with `Maybe` to express *optional results* in the context of some global configuration. Or we might want the mutable state provided by the `State` monad together with the pure error tracking capability of the `Either` monad. This is the problem solved by *monad transformers*.

Note that we have already seen that the `Effect` monad partially solves this problem. Monad transformers provide another solution, and each approach has its own benefits and limitations.

A monad transformer is a type constructor parameterized by a type and another type constructor. It takes one monad and turns it into another monad, adding its own variety of side-effects.

Let's see an example. The monad transformer version of the `State` monad is `StateT`, defined in the `Control.Monad.State.Trans` module. We can find the kind of `StateT` using PSCi:

```
> import Control.Monad.State.Trans
> :kind StateT
Type -> (Type -> Type) -> Type -> Type
```

This looks quite confusing, but we can apply `StateT` one argument at a time to understand how to use it.

The first type argument is the type of the state we wish to use, as was the case for `State`. Let's use a state of type `String`:

```
> :kind StateT String
(Type -> Type) -> Type -> Type
```

The next argument is a type constructor of kind `Type -> Type`. It represents the underlying monad, which we want to add the effects of `StateT` to. For the sake of an example, let's choose the `Either String` monad:

```
> :kind StateT String (Either String)
Type -> Type
```

We are left with a type constructor. The final argument represents the return type, and we might instantiate it to `Number` for example:

```
> :kind StateT String (Either String) Number
Type
```

Finally, we are left with something of kind `Type`, which means we can try to find values of this type.

The monad we have constructed – `StateT String (Either String)` – represents computations that can fail with an error and use mutable state.

We can use the actions of the outer `StateT String` monad (`get`, `put`, and `modify`) directly, but to use the effects of the wrapped monad (`Either String`), we need to "lift" them over the monad transformer. The `Control.Monad.Trans` module defines the `MonadTrans` type class, which captures those type constructors which are monad transformers, as follows:

```
class MonadTrans t where
  lift :: forall m a. Monad m => m a -> t m a
```

This class contains a single member, `lift`, which takes computations in any underlying

monad `m` and lifts them into the wrapped monad `t m`. In our case, the type constructor `t` is `StateT String`, `m` is the `Either String` monad, so `lift` provides a way to lift computations of type `Either String a` to computations of type `StateT String (Either String) a`. This means that we can use the effects of `StateT String` and `Either String` side-by-side, as long as we use `lift` every time we use a computation of type `Either String a`.

For example, the following computation reads the underlying state and then throws an error if the state is the empty string:

```
import Data.String (drop, take)

split :: StateT String (Either String) String
split = do
  s <- get
  case s of
    "" -> lift $ Left "Empty string"
    _ -> do
      put (drop 1 s)
      pure (take 1 s)
```

If the state is not empty, the computation uses `put` to update the state to `drop 1 s` (that is, `s` with the first character removed) and returns `take 1 s` (that is, the first character of `s`).

Let's try this in PSCi:

```
> runStateT split "test"
Right (Tuple "t" "est")

> runStateT split ""
Left "Empty string"
```

This is not very remarkable since we could have implemented this without `StateT`. However, since we are working in a monad, we can use do notation or applicative combinators to build larger computations from smaller ones. For example, we can apply `split` twice to read the first two characters from a string:

```
> runStateT ((<>) <$> split <*> split) "test"
(Right (Tuple "te" "st"))
```

We can use the `split` function with a handful of other actions to build a basic parsing library. In fact, this is the approach taken by the `parsing` library. This is the power of monad transformers – we can create custom-built monads for various problems, choose the side-effects we need, and keep the expressiveness of do notation and applicative combinators.

The ExceptT Monad Transformer

The `transformers` package also defines the `ExceptT e` monad transformer, corresponding to the `Either e` monad. It provides the following API:

```
class MonadError e m where
  throwError :: forall a. e -> m a
  catchError :: forall a. m a -> (e -> m a) -> m a

instance Monad m => MonadError e (ExceptT e m)

runExceptT :: forall e m a. ExceptT e m a -> m (Either e a)
```

The `MonadError` class captures those monads that support throwing and catching errors of some type `e`, and an instance is provided for the `ExceptT e` monad transformer. The `throwError` action can indicate failure, like `Left` in the `Either e` monad. The `catchError` action allows us to continue after an error is thrown using `throwError`.

The `runExceptT` handler is used to run a computation of type `ExceptT e m a`.

This API is similar to that provided by the `exceptions` package and the `Exception` effect. However, there are some important differences:

- `Exception` uses actual JavaScript exceptions, whereas `ExceptT` models errors as a pure data structure.
- The `Exception` effect only supports exceptions of one type, namely JavaScript's `Error` type, whereas `ExceptT` supports errors of any type. In particular, we are free to define new error types.

Let's try out `ExceptT` by using it to wrap the `Writer` monad. Again, we are free to use actions from the monad transformer `ExceptT e` directly, but computations in the `Writer` monad should be lifted using `lift`:

```
import Control.Monad.Except
import Control.Monad.Writer

writerAndExceptT :: ExceptT String (Writer (Array String)) String
writerAndExceptT = do
  lift $ tell ["Before the error"]
  _ <- throwError "Error!"
  lift $ tell ["After the error"]
  pure "Return value"
```

If we test this function in PSCi, we can see how the two effects of accumulating a log and

throwing an error interact. First, we can run the outer `ExceptT` computation of type by using `runExceptT`, leaving a result of type `Writer (Array String) (Either String String)`. We can then use `runWriter` to run the inner `Writer` computation:

```
> runWriter $ runExceptT writerAndExceptT
Tuple (Left "Error!") ["Before the error"]
```

Note that only those log messages that were written before the error was thrown get appended to the log.

Monad Transformer Stacks

As we have seen, monad transformers can be used to build new monads on top of existing monads. For some monad transformer `t1` and some monad `m`, the application `t1 m` is also a monad. That means we can apply a *second* monad transformer `t2` to the result `t1 m` to construct a third monad `t2 (t1 m)`. In this way, we can construct a *stack* of monad transformers, which combine the side-effects provided by their constituent monads.

In practice, the underlying monad `m` is either the `Effect` monad, if native side-effects are required, or the `Identity` monad, defined in the `Data.Identity` module. The `Identity` monad adds no new side-effects, so transforming the `Identity` monad only provides the effects of the monad transformer. The `State`, `Reader`, and `Writer` monads are implemented by transforming the `Identity` monad with `StateT`, `ReaderT`, and `WriterT`, respectively.

Let's see an example in which three side effects are combined. We will use the `StateT`, `WriterT`, and `ExceptT` effects, with the `Identity` monad on the bottom of the stack. This monad transformer stack will provide the side effects of mutable state, accumulating a log, and pure errors.

We can use this monad transformer stack to reproduce our `split` action with the added feature of logging.

```
type Errors = Array String

type Log = Array String

type Parser = StateT String (WriterT Log (ExceptT Errors Identity))

split :: Parser String
split = do
  s <- get
  lift $ tell ["The state is " <> s]
  case s of
    "" -> lift $ lift $ throwError ["Empty string"]
    _ -> do
      put (drop 1 s)
      pure (take 1 s)
```

If we test this computation in PSCi, we see that the state is appended to the log for every invocation of `split`.

Note that we have to remove the side-effects in the order in which they appear in the monad transformer stack: first, we use `runStateT` to remove the `StateT` type constructor, then `runWriterT`, then `runExceptT`. Finally, we run the computation in the `Identity` monad by using `unwrap`.

```
> runParser p s = unwrap $ runExceptT $ runWriterT $ runStateT p s

> runParser split "test"
(Right (Tuple (Tuple "t" "est") ["The state is test"]))

> runParser ((<>) <$> split <*> split) "test"
(Right (Tuple (Tuple "te" "st") ["The state is test", "The state is est"]))
```

However, if the parse is unsuccessful because the state is empty, then no log is printed at all:

```
> runParser split ""
(Left ["Empty string"])
```

This is because of how the side-effects provided by the `ExceptT` monad transformer interact with the side-effects provided by the `WriterT` monad transformer. We can address this by changing the order in which the monad transformer stack is composed. If we move the `ExceptT` transformer to the top of the stack, then the log will contain all messages written up until the first error, as we saw earlier when we transformed `Writer` with `ExceptT`.

One problem with this code is that we have to use the `lift` function multiple times to lift

computations over multiple monad transformers; for example, the call to `throwError` has to be lifted twice, once over `WriterT` and a second time over `StateT`. This is fine for small monad transformer stacks but quickly becomes inconvenient.

Fortunately, as we will see, we can use the automatic code generation provided by type class inference to do most of this "heavy lifting" for us.

Exercises

1. (Easy) Use the `ExceptT` monad transformer over the `Identity` functor to write a function `safeDivide` which divides two numbers, throwing an error (as the String "Divide by zero!") if the denominator is zero.

2. (Medium) Write a parser

   ```
   string :: String -> Parser String
   ```

   which matches a string as a prefix of the current state or fails with an error message.

   Your parser should work as follows:

   ```
   > runParser (string "abc") "abcdef"
   (Right (Tuple (Tuple "abc" "def") ["The state is abcdef"]))
   ```

   Hint: you can use the implementation of `split` as a starting point. You might find the `stripPrefix` function useful.

3. (Difficult) Use the `ReaderT` and `WriterT` monad transformers to reimplement the document printing library, which we wrote earlier using the `Reader` monad.

   Instead of using `line` to emit strings and `cat` to concatenate strings, use the `Array String` monoid with the `WriterT` monad transformer, and `tell` to append a line to the result. Use the same names as in the original implementation but ending with an apostrophe (`'`).

Type Classes to the Rescue!

When we looked at the `State` monad at the start of this chapter, I gave the following types

for the actions of the `State` monad:

```
get    :: forall s.            State s s
put    :: forall s. s       -> State s Unit
modify :: forall s. (s -> s) -> State s Unit
```

In reality, the types given in the `Control.Monad.State.Class` module are more general than this:

```
get    :: forall m s. MonadState s m =>            m s
put    :: forall m s. MonadState s m => s       -> m Unit
modify :: forall m s. MonadState s m => (s -> s) -> m Unit
```

The `Control.Monad.State.Class` module defines the `MonadState` (multi-parameter) type class, which allows us to abstract over "monads which support pure mutable state". As one would expect, the `State s` type constructor is an instance of the `MonadState s` type class, but there are many more interesting instances of this class.

In particular, there are instances of `MonadState` for the `WriterT`, `ReaderT`, and `ExceptT` monad transformers provided in the `transformers` package. Each has an instance for `MonadState` whenever the underlying `Monad` does. In practice, this means that as long as `StateT` appears *somewhere* in the monad transformer stack, and everything above `StateT` is an instance of `MonadState`, then we are free to use `get`, `put`, and `modify` directly without the need to use `lift`.

Indeed, the same is true of the actions we covered for the `ReaderT`, `WriterT`, and `ExceptT` transformers. `transformers` defines a type class for each of the major transformers, allowing us to abstract over monads that support their operations.

In the case of the `split` function above, the monad stack we constructed is an instance of each of the `MonadState`, `MonadWriter`, and `MonadError` type classes. This means that we don't need to call `lift` at all! We can just use the actions `get`, `put`, `tell`, and `throwError` as if they were defined on the monad stack itself:

```
split :: Parser String
split = do
  s <- get
  tell ["The state is " <> show s]
  case s of
    "" -> throwError ["Empty string"]
    _ -> do
      put (drop 1 s)
      pure (take 1 s)
```

This computation looks like we have extended our programming language to support the three new side-effects of mutable state, logging, and error handling. However, everything is still implemented using pure functions and immutable data under the hood.

Alternatives

The `control` package defines a number of abstractions for working with computations that can fail. One of these is the `Alternative` type class:

```
class Functor f <= Alt f where
  alt :: forall a. f a -> f a -> f a

class Alt f <= Plus f where
  empty :: forall a. f a

class (Applicative f, Plus f) <= Alternative f
```

`Alternative` provides two new combinators: the `empty` value, which provides a prototype for a failing computation, and the `alt` function (and its alias, `<|>`), which provides the ability to fall back to an *alternative* computation in the case of an error.

The `Data.Array` module provides two useful functions for working with type constructors in the `Alternative` type class:

```
many :: forall f a. Alternative f => Lazy (f (Array a)) => f a -> f (Array a)
some :: forall f a. Alternative f => Lazy (f (Array a)) => f a -> f (Array a)
```

There is also an equivalent `many` and `some` for `Data.List`

The `many` combinator uses the `Alternative` type class to repeatedly run a computation *zero-or-more* times. The `some` combinator is similar but requires at least the first computation to succeed.

In the case of our `Parser` monad transformer stack, there is an instance of `Alternative` induced by the `ExceptT` component, which supports failure by composing errors in different branches using a `Monoid` instance (this is why we chose `Array String` for our `Errors` type). This means that we can use the `many` and `some` functions to run a parser multiple times:

```
> import Data.Array (many)

> runParser (many split) "test"
(Right (Tuple (Tuple ["t", "e", "s", "t"] "")
              [ "The state is \"test\""
              , "The state is \"est\""
              , "The state is \"st\""
              , "The state is \"t\""
              ]))
```

Here, the input string `"test"` has been repeatedly split to return an array of four single-character strings, the leftover state is empty, and the log shows that we applied the `split` combinator four times.

Monad Comprehensions

The `Control.MonadPlus` module defines a subclass of the `Alternative` type class, called `MonadPlus`. `MonadPlus` captures those type constructors which are both monads and instances of `Alternative`:

```
class (Monad m, Alternative m) <= MonadPlus m
```

In particular, our `Parser` monad is an instance of `MonadPlus`.

When we covered array comprehensions earlier in the book, we introduced the `guard` function, which could be used to filter out unwanted results. In fact, the `guard` function is more general and can be used for any monad, which is an instance of `MonadPlus`:

```
guard :: forall m. Alternative m => Boolean -> m Unit
```

The `<|>` operator allows us to backtrack in case of failure. To see how this is useful, let's define a variant of the `split` combinator which only matches upper case characters:

```
upper :: Parser String
upper = do
  s <- split
  guard $ toUpper s == s
  pure s
```

Here, we use a `guard` to fail if the string is not upper case. Note that this code looks very similar to the array comprehensions we saw earlier – using `MonadPlus` in this way, we

sometimes refer to constructing *monad comprehensions*.

Backtracking

We can use the `<|>` operator to backtrack to another alternative in case of failure. To demonstrate this, let's define one more parser, which matches lower case characters:

```
lower :: Parser String
lower = do
  s <- split
  guard $ toLower s == s
  pure s
```

With this, we can define a parser which eagerly matches many upper case characters if the first character is upper case, or many lower case character if the first character is lower case:

```
> upperOrLower = some upper <|> some lower
```

This parser will match characters until the case changes:

```
> runParser upperOrLower "abcDEF"
(Right (Tuple (Tuple ["a","b","c"] ("DEF"))
              [ "The state is \"abcDEF\""
              , "The state is \"bcDEF\""
              , "The state is \"cDEF\""
              ]))
```

We can even use `many` to fully split a string into its lower and upper case components:

```
> components = many upperOrLower

> runParser components "abCDeFgh"
(Right (Tuple (Tuple [["a","b"],["C","D"],["e"],["F"],["g","h"]] "")
              [ "The state is \"abCDeFgh\""
              , "The state is \"bCDeFgh\""
              , "The state is \"CDeFgh\""
              , "The state is \"DeFgh\""
              , "The state is \"eFgh\""
              , "The state is \"Fgh\""
              , "The state is \"gh\""
              , "The state is \"h\""
              ]))
```

Again, this illustrates the power of reusability that monad transformers bring – we were able

to write a backtracking parser in a declarative style with only a few lines of code, by reusing standard abstractions!

Exercises

1. (Easy) Remove the calls to the `lift` function from your implementation of the `string` parser. Verify that the new implementation type checks, and convince yourself it should.
2. (Medium) Use your `string` parser with the `some` combinator to write a parser `asFollowedByBs` that recognizes strings consisting of several copies of the string `"a"` followed by several copies of the string `"b"`.
3. (Medium) Use the `<|>` operator to write a parser `asOrBs` that recognizes strings of the letters `a` or `b` in any order.
4. (Difficult) The `Parser` monad might also be defined as follows:

   ```
   type Parser = ExceptT Errors (StateT String (WriterT Log Identity))
   ```

   What effect does this change have on our parsing functions?

The RWS Monad

One particular combination of monad transformers is so common that it is provided as a single monad transformer in the `transformers` package. The `Reader`, `Writer`, and `State` monads are combined into the *reader-writer-state* or simply `RWS` monad. This monad has a corresponding monad transformer called the `RWST` monad transformer.

We will use the `RWS` monad to model the game logic for our text adventure game.

The `RWS` monad is defined in terms of three type parameters (in addition to its return type):

```
type RWS r w s = RWST r w s Identity
```

Notice that the `RWS` monad is defined as its own monad transformer by setting the base monad to `Identity`, which provides no side-effects.

The first type parameter, `r`, represents the global configuration type. The second, `w`, represents the monoid, which we will use to accumulate a log, and the third, `s`, is the type of our mutable state.

In the case of our game, our global configuration is defined in a type called `GameEnvironment` in the `Data.GameEnvironment` module:

```
type PlayerName = String

newtype GameEnvironment = GameEnvironment
  { playerName    :: PlayerName
  , debugMode     :: Boolean
  }
```

It defines the player name and a flag that indicates whether or not the game is running in debug mode. These options will be set from the command line when we come to run our monad transformer.

The mutable state is defined in a type called `GameState` in the `Data.GameState` module:

```
import Data.Map as M
import Data.Set as S

newtype GameState = GameState
  { items       :: M.Map Coords (S.Set GameItem)
  , player      :: Coords
  , inventory   :: S.Set GameItem
  }
```

The `Coords` data type represents points on a two-dimensional grid, and the `GameItem` data type is an enumeration of the items in the game:

```
data GameItem = Candle | Matches
```

The `GameState` type uses two new data structures: `Map` and `Set`, which represent sorted maps and sorted sets, respectively. The `items` property is a mapping from coordinates of the game grid to sets of game items at that location. The `player` property stores the current coordinates of the player, and the `inventory` property stores a set of game items currently held by the player.

The `Map` and `Set` data structures are sorted by their keys, and can be used with any key type in the `Ord` type class. This means that the keys in our data structures should be totally ordered.

We will see how the `Map` and `Set` structures are used as we write the actions for our game.

For our log, we will use the `List String` monoid. We can define a type synonym for our `Game` monad, implemented using `RWS`:

```
type Log = L.List String

type Game = RWS GameEnvironment Log GameState
```

Implementing Game Logic

Our game will be built from simple actions defined in the `Game` monad by reusing the actions from the `Reader`, `Writer`, and `State` monads. At the top level of our application, we will run the pure computations in the `Game` monad and use the `Effect` monad to turn the results into observable side-effects, such as printing text to the console.

One of the simplest actions in our game is the `has` action. This action tests whether the player's inventory contains a particular game item. It is defined as follows:

```
has :: GameItem -> Game Boolean
has item = do
  GameState state <- get
  pure $ item `S.member` state.inventory
```

This function uses the `get` action defined in the `MonadState` type class to read the current game state and then uses the `member` function defined in `Data.Set` to test whether the specified `GameItem` appears in the `Set` of inventory items.

Another action is the `pickUp` action. It adds a game item to the player's inventory if it appears in the current room. It uses actions from the `MonadWriter` and `MonadState` type classes. First of all, it reads the current game state:

```
pickUp :: GameItem -> Game Unit
pickUp item = do
  GameState state <- get
```

Next, `pickUp` looks up the set of items in the current room. It does this by using the `lookup` function defined in `Data.Map`:

```
  case state.player `M.lookup` state.items of
```

The `lookup` function returns an optional result indicated by the `Maybe` type constructor. If the key does not appear in the map, the `lookup` function returns `Nothing`; otherwise, it returns the corresponding value in the `Just` constructor.

We are interested in the case where the corresponding item set contains the specified game item. Again we can test this using the `member` function:

```
    Just items | item `S.member` items -> do
```

In this case, we can use `put` to update the game state and `tell` to add a message to the log:

```
          let newItems = M.update (Just <<< S.delete item) state.player
state.items
              newInventory = S.insert item state.inventory
          put $ GameState state { items     = newItems
                                , inventory = newInventory
                                }
          tell (L.singleton ("You now have the " <> show item))
```

Note that there is no need to `lift` either of the two computations here because there are appropriate instances for both `MonadState` and `MonadWriter` for our `Game` monad transformer stack.

The argument to `put` uses a record update to modify the game state's `items` and `inventory` fields. We use the `update` function from `Data.Map`, which modifies a value at a particular key. In this case, we modify the set of items at the player's current location, using the `delete` function to remove the specified item from the set. `inventory` is also updated, using `insert` to add the new item to the player's inventory set.

Finally, the `pickUp` function handles the remaining cases by notifying the user using `tell`:

```
    _ -> tell (L.singleton "I don't see that item here.")
```

As an example of using the `Reader` monad, we can look at the code for the `debug` command. This command allows the user to inspect the game state at runtime if the game is running in debug mode:

```
  GameEnvironment env <- ask
  if env.debugMode
    then do
      state :: GameState <- get
      tell (L.singleton (show state))
    else tell (L.singleton "Not running in debug mode.")
```

Here, we use the `ask` action to read the game configuration. Again, note that we don't need to `lift` any computation, and we can use actions defined in the `MonadState`, `MonadReader`, and `MonadWriter` type classes in the same do notation block.

If the `debugMode` flag is set, the `tell` action is used to write the state to the log. Otherwise, an error message is added.

The remainder of the `Game` module defines a set of similar actions, each using only the actions defined by the `MonadState`, `MonadReader`, and `MonadWriter` type classes.

Running the Computation

Since our game logic runs in the `RWS` monad, it is necessary to run the computation to respond to the user's commands.

The front-end of our game is built using two packages: `optparse`, which provides applicative command line parsing, and `node-readline`, which wraps NodeJS' `readline` module, allowing us to write interactive console-based applications.

The interface to our game logic is provided by the function `game` in the `Game` module:

```
game :: Array String -> Game Unit
```

To run this computation, we pass a list of words entered by the user as an array of strings and run the resulting `RWS` computation using `runRWS`:

```
data RWSResult state result writer = RWSResult state result writer

runRWS :: forall r w s a. RWS r w s a -> r -> s -> RWSResult s a w
```

`runRWS` looks like a combination of `runReader`, `runWriter`, and `runState`. It takes a global configuration and an initial state as an argument and returns a data structure containing the log, the result, and the final state.

The front-end of our application is defined by a function `runGame`, with the following type signature:

```
runGame :: GameEnvironment -> Effect Unit
```

This function interacts with the user via the console (using the `node-readline` and `console` packages). `runGame` takes the game configuration as a function argument.

The `node-readline` package provides the `LineHandler` type, which represents actions in the `Effect` monad, which handle user input from the terminal. Here is the corresponding API:

```
type LineHandler a = String -> Effect a

foreign import setLineHandler
  :: forall a
   . Interface
  -> LineHandler a
  -> Effect Unit
```

The `Interface` type represents a handle for the console and is passed as an argument to the functions which interact with it. An `Interface` can be created using the `createConsoleInterface` function:

```
import Node.ReadLine as RL

runGame env = do
  interface <- RL.createConsoleInterface RL.noCompletion
```

The first step is to set the prompt at the console. We pass the `interface` handle, and provide the prompt string and indentation level:

```
  RL.setPrompt "> " interface
```

In our case, we are interested in implementing the line handler function. Our line handler is defined using a helper function in a `let` declaration, as follows:

```
lineHandler :: GameState -> String -> Effect Unit
lineHandler currentState input = do
  case runRWS (game (split (wrap " ") input)) env currentState of
    RWSResult state _ written -> do
      for_ written log
      RL.setLineHandler (lineHandler state) $ interface
  RL.prompt interface
  pure unit
```

The `let` binding is closed over both the game configuration, named `env`, and the console handle, named `interface`.

Our handler takes an additional first argument, the game state. This is required since we need to pass the game state to `runRWS` to run the game's logic.

The first thing this action does is to break the user input into words using the `split` function from the `Data.String` module. It then uses `runRWS` to run the `game` action (in the `RWS` monad), passing the game environment and current game state.

Having run the game logic, which is a pure computation, we need to print any log messages to the screen and show the user a prompt for the next command. The `for_` action is used to traverse the log (of type `List String`) and print its entries to the console. Finally, `setLineHandler` is used to update the line handler function to use the updated game state, and the prompt is displayed again using the `prompt` action.

The `runGame` function finally attaches the initial line handler to the console interface and displays the initial prompt:

```
RL.setLineHandler (lineHandler initialGameState) interface
RL.prompt interface
```

Exercises

1. (Medium) Implement a new command `cheat`, which moves all game items from the game grid into the user's inventory. Create a function `cheat :: Game Unit` in the `Game` module, and use this function from `game`.

2. (Difficult) The `Writer` component of the `RWS` monad is currently used for two types of messages: error messages and informational messages. Because of this, several parts of the code use case statements to handle error cases.

Refactor the code to use the `ExceptT` monad transformer to handle the error messages and `RWS` to handle informational messages. *Note:* There are no tests for this exercise.

Handling Command Line Options

The final piece of the application is responsible for parsing command line options and creating the `GameEnvironment` configuration record. For this, we use the `optparse` package.

`optparse` is an example of *applicative command line option parsing*. Recall that an applicative functor allows us to lift functions of arbitrary arity over a type constructor representing some type of side-effect. In the case of the `optparse` package, the functor we are interested in is the `Parser` functor (imported from the optparse module `Options.Applicative`, not to be confused with our `Parser` that we defined in the `Split` module), which adds the side-effect of reading from command line options. It provides the following handler:

```
customExecParser :: forall a. ParserPrefs -> ParserInfo a -> Effect a
```

This is best illustrated by example. The application's `main` function is defined using `customExecParser` as follows:

```
main = OP.customExecParser prefs argParser >>= runGame
```

The first argument is used to configure the `optparse` library. In our case, we simply configure it to show the help message when the application is run without any arguments (instead of showing a "missing argument" error) by using `OP.prefs OP.showHelpOnEmpty`, but the `Options.Applicative.Builder` module provides several other options.

The second argument is the complete description of our parser program:

```
argParser :: OP.ParserInfo GameEnvironment
argParser = OP.info (env <**> OP.helper) parserOptions

parserOptions = fold
  [ OP.fullDesc
  , OP.progDesc "Play the game as <player name>"
  , OP.header "Monadic Adventures! A game to learn monad transformers"
  ]
```

Here `OP.info` combines a `Parser` with a set of options for how the help message is formatted. `env <**> OP.helper` takes any command line argument `Parser` named `env`

and automatically adds a `--help` option. Options for the help message are of type `InfoMod`, which is a monoid, so we can use the `fold` function to add several options together.

The interesting part of our parser is constructing the `GameEnvironment`:

```
env :: OP.Parser GameEnvironment
env = gameEnvironment <$> player <*> debug

player :: OP.Parser String
player = OP.strOption $ fold
  [ OP.long "player"
  , OP.short 'p'
  , OP.metavar "<player name>"
  , OP.help "The player's name <String>"
  ]

debug :: OP.Parser Boolean
debug = OP.switch $ fold
  [ OP.long "debug"
  , OP.short 'd'
  , OP.help "Use debug mode"
  ]
```

`player` and `debug` are both `Parser`s, so we can use our applicative operators `<$>` and `<*>` to lift our `gameEnvironment` function, which has the type `PlayerName -> Boolean -> GameEnvironment` over `Parser`. `OP.strOption` constructs a command line option that expects a string value and is configured via a collection of `Mod`s folded together. `OP.flag` works similarly but doesn't expect an associated value. `optparse` offers extensive documentation on different modifiers available to build various command line parsers.

Notice how we used the notation afforded by the applicative operators to give a compact, declarative specification of our command line interface. In addition, it is simple to add new command line arguments by adding a new function argument to `runGame` and then using `<*>` to lift `runGame` over an additional argument in the definition of `env`.

Exercises

1. (Medium) Add a new Boolean-valued property `cheatMode` to the `GameEnvironment` record. Add a new command line flag `-c` to the `optparse` configuration, enabling cheat mode. The `cheat` command from the previous exercise should be disallowed if cheat mode is not enabled.

Conclusion

This chapter was a practical demonstration of the techniques we've learned so far, using monad transformers to build a pure specification of our game and the `Effect` monad to build a front-end using the console.

Because we separated our implementation from the user interface, it would be possible to create other front-ends for our game. For example, we could use the `Effect` monad to render the game in the browser using the Canvas API or the DOM.

We have seen how monad transformers allow us to write safe code in an imperative style, where the type system tracks effects. In addition, type classes provide a powerful way to abstract over the actions provided by a monad, enabling code reuse. We used standard abstractions like `Alternative` and `MonadPlus` to build useful monads by combining standard monad transformers.

Monad transformers are an excellent demonstration of expressive code that can be written by relying on advanced type system features such as higher-kinded polymorphism and multi-parameter type classes.

Canvas Graphics

Chapter Goals

This chapter will be an extended example focussing on the `canvas` package, which provides a way to generate 2D graphics from PureScript using the HTML5 Canvas API.

Project Setup

This module's project introduces the following new dependencies:

- `canvas`, which gives types to methods from the HTML5 Canvas API
- `refs`, which provides a side-effect for using *global mutable references*

The source code for the chapter is broken up into a set of modules, each of which defines a `main` method. Different sections of this chapter are implemented in different files, and the `Main` module can be changed by modifying the Spago build command to run the appropriate file's `main` method at each point.

The HTML file `html/index.html` contains a single `canvas` element which will be used in each example, and a `script` element to load the compiled PureScript code. To test the code for each section, open the HTML file in your browser. Because most exercises target the browser, this chapter has no unit tests.

Simple Shapes

The `Example/Rectangle.purs` file contains a simple introductory example, which draws a single blue rectangle at the center of the canvas. The module imports the `Effect` type from the `Effect` module, and also the `Graphics.Canvas` module, which contains actions in the `Effect` monad for working with the Canvas API.

The `main` action starts, like in the other modules, by using the `getCanvasElementById` action to get a reference to the canvas object and the `getContext2D` action to access the 2D rendering context for the canvas:

The `void` function takes a functor and replaces its value with `Unit`. In the example, it is used to make `main` conform with its signature.

```
main :: Effect Unit
main = void $ unsafePartial do
  Just canvas <- getCanvasElementById "canvas"
  ctx <- getContext2D canvas
```

Note: the call to `unsafePartial` here is necessary since the pattern match on the result of `getCanvasElementById` is partial, matching only the `Just` constructor. For our purposes, this is fine, but in production code, we would probably want to match the `Nothing` constructor and provide an appropriate error message.

The types of these actions can be found using PSCi or by looking at the documentation:

```
getCanvasElementById :: String -> Effect (Maybe CanvasElement)

getContext2D :: CanvasElement -> Effect Context2D
```

`CanvasElement` and `Context2D` are types defined in the `Graphics.Canvas` module. The same module also defines the `Canvas` effect, which is used by all of the actions in the module.

The graphics context `ctx` manages the state of the canvas and provides methods to render primitive shapes, set styles and colors, and apply transformations.

We continue by setting the fill style to solid blue using the `setFillStyle` action. The longer hex notation of `#0000FF` may also be used for blue, but shorthand notation is easier for simple colors:

```
  setFillStyle ctx "#00F"
```

Note that the `setFillStyle` action takes the graphics context as an argument. This is a common pattern in the `Graphics.Canvas` module.

Finally, we use the `fillPath` action to fill the rectangle. `fillPath` has the following type:

```
fillPath :: forall a. Context2D -> Effect a -> Effect a
```

`fillPath` takes a graphics context and another action that builds the path to render. To build a path, we can use the `rect` action. `rect` takes a graphics context and a record that provides the position and size of the rectangle:

```
  fillPath ctx $ rect ctx
    { x: 250.0
    , y: 250.0
    , width: 100.0
    , height: 100.0
    }
```

Build the rectangle example, providing `Example.Rectangle` as the name of the main module:

```
$ spago bundle-app --main Example.Rectangle --to dist/Main.js
```

Now, open the `html/index.html` file and verify that this code renders a blue rectangle in the center of the canvas.

Putting Row Polymorphism to Work

There are other ways to render paths. The `arc` function renders an arc segment, and the `moveTo`, `lineTo`, and `closePath` functions can render piecewise-linear paths.

The `Shapes.purs` file renders three shapes: a rectangle, an arc segment, and a triangle.

We have seen that the `rect` function takes a record as its argument. In fact, the properties of the rectangle are defined in a type synonym:

```
type Rectangle =
  { x :: Number
  , y :: Number
  , width :: Number
  , height :: Number
  }
```

The `x` and `y` properties represent the location of the top-left corner, while the `width` and `height` properties represent the lengths of the rectangle, respectively.

To render an arc segment, we can use the `arc` function, passing a record with the following type:

```
type Arc =
  { x      :: Number
  , y      :: Number
  , radius :: Number
  , start  :: Number
  , end    :: Number
  }
```

Here, the `x` and `y` properties represent the center point, `radius` is the radius, `start` and `end` represent the endpoints of the arc in radians.

For example, this code fills an arc segment centered at `(300, 300)` with radius `50`. The arc completes 2/3rds of a rotation. Note that the unit circle is flipped vertically since the y-axis increases towards the bottom of the canvas:

```
fillPath ctx $ arc ctx
  { x      : 300.0
  , y      : 300.0
  , radius : 50.0
  , start  : 0.0
  , end    : Math.tau * 2.0 / 3.0
  }
```

Notice that both the `Rectangle` and `Arc` record types contain `x` and `y` properties of type `Number`. In both cases, this pair represents a point. This means we can write row-polymorphic functions acting on either type of record.

For example, the `Shapes` module defines a `translate` function that translates a shape by modifying its `x` and `y` properties:

```
translate
  :: forall r
   . Number
  -> Number
  -> { x :: Number, y :: Number | r }
  -> { x :: Number, y :: Number | r }
translate dx dy shape = shape
  { x = shape.x + dx
  , y = shape.y + dy
  }
```

Notice the row-polymorphic type. It says that `translate` accepts any record with `x` and `y` properties *and any other properties*, and returns the same type of record. The `x` and `y` fields are updated, but the rest of the fields remain unchanged.

This is an example of *record update syntax*. The expression `shape { ... }` creates a new

record based on the `shape` record, with the fields inside the braces updated to the specified values. Note that the expressions inside the braces are separated from their labels by equals symbols, not colons like in record literals.

The `translate` function can be used with both the `Rectangle` and `Arc` records, as seen in the `Shapes` example.

The third type of path rendered in the `Shapes` example is a piecewise-linear path. Here is the corresponding code:

```
setFillStyle ctx "#F00"

fillPath ctx $ do
  moveTo ctx 300.0 260.0
  lineTo ctx 260.0 340.0
  lineTo ctx 340.0 340.0
  closePath ctx
```

There are three functions in use here:

- `moveTo` moves the current location of the path to the specified coordinates,
- `lineTo` renders a line segment between the current location and the specified coordinates, and updates the current location,
- `closePath` completes the path by rendering a line segment joining the current location to the start position.

The result of this code snippet is to fill an isosceles triangle.

Build the example by specifying `Example.Shapes` as the main module:

```
$ spago bundle-app --main Example.Shapes --to dist/Main.js
```

and open `html/index.html` again to see the result. You should see the three different types of shapes rendered to the canvas.

Exercises

1. (Easy) Experiment with the `strokePath` and `setStrokeStyle` functions in each example so far.

2. (Easy) The `fillPath` and `strokePath` functions can render complex paths with a common style using a do notation block inside the function argument. Try changing the

`Rectangle` example to render two rectangles side-by-side using the same call to `fillPath`. Try rendering a sector of a circle by using a combination of a piecewise-linear path and an arc segment.

3. (Medium) Given the following record type:

```
type Point = { x :: Number, y :: Number }
```

which represents a 2D point, write a function `renderPath` which strokes a closed path constructed from a number of points:

```
renderPath
  :: Context2D
  -> Array Point
  -> Effect Unit
```

Given a function

```
f :: Number -> Point
```

which takes a `Number` between `0` and `1` as its argument and returns a `Point`, write an action that plots `f` by using your `renderPath` function. Your action should approximate the path by sampling `f` at a finite set of points.

Experiment by rendering different paths by varying the function `f`.

Drawing Random Circles

The `Example/Random.purs` file contains an example that uses the `Effect` monad to interleave two types of side-effect: random number generation and canvas manipulation. The example renders one hundred randomly generated circles onto the canvas.

The `main` action obtains a reference to the graphics context as before and then sets the stroke and fill styles:

```
setFillStyle ctx "#F00"
setStrokeStyle ctx "#000"
```

Next, the code uses the `for_` function to loop over the integers between `0` and `100`:

```
  for_ (1 .. 100) \_ -> do
```

On each iteration, the do notation block starts by generating three random numbers distributed between `0` and `1`. These numbers represent the `x` and `y` coordinates and the radius of a circle:

```
    x <- random
    y <- random
    r <- random
```

Next, for each circle, the code creates an `Arc` based on these parameters and finally fills and strokes the arc with the current styles:

```
    let path = arc ctx
          { x                   : x * 600.0
          , y                   : y * 600.0
          , radius              : r * 50.0
          , start               : 0.0
          , end                 : Number.tau
          , useCounterClockwise: false
          }

    fillPath ctx path
    strokePath ctx path
```

Build this example by specifying the `Example.Random` module as the main module:

```
$ spago bundle-app --main Example.Random --to dist/Main.js
```

and view the result by opening `html/index.html`.

Transformations

There is more to the canvas than just rendering simple shapes. Every canvas maintains a transformation that is used to transform shapes before rendering. Shapes can be translated, rotated, scaled, and skewed.

The `canvas` library supports these transformations using the following functions:

```
translate :: Context2D
          -> TranslateTransform
          -> Effect Context2D

rotate    :: Context2D
          -> Number
          -> Effect Context2D

scale     :: Context2D
          -> ScaleTransform
          -> Effect Context2D

transform :: Context2D
          -> Transform
          -> Effect Context2D
```

The `translate` action performs a translation whose components are specified by the properties of the `TranslateTransform` record.

The `rotate` action rotates around the origin through some number of radians specified by the first argument.

The `scale` action performs a scaling, with the origin as the center. The `ScaleTransform` record specifies the scale factors along the `x` and `y` axes.

Finally, `transform` is the most general action of the four here. It performs an affine transformation specified by a matrix.

Any shapes rendered after these actions have been invoked will automatically have the appropriate transformation applied.

In fact, the effect of each of these functions is to *post-multiply* the transformation with the context's current transformation. The result is that if multiple transformations applied after one another, then their effects are actually applied in reverse:

```
transformations ctx = do
  translate ctx { translateX: 10.0, translateY: 10.0 }
  scale ctx { scaleX: 2.0, scaleY: 2.0 }
  rotate ctx (Math.tau / 4.0)

  renderScene
```

The effect of this sequence of actions is that the scene is rotated, then scaled, and finally translated.

Preserving the Context

A common use case is to render some subset of the scene using a transformation and then reset the transformation.

The Canvas API provides the `save` and `restore` methods, which manipulate a *stack* of states associated with the canvas. `canvas` wraps this functionality into the following functions:

```
save
  :: Context2D
  -> Effect Context2D

restore
  :: Context2D
  -> Effect Context2D
```

The `save` action pushes the current state of the context (including the current transformation and any styles) onto the stack, and the `restore` action pops the top state from the stack and restores it.

This allows us to save the current state, apply some styles and transformations, render some primitives, and finally restore the original transformation and state. For example, the following function performs some canvas action but applies a rotation before doing so and restores the transformation afterwards:

```
rotated ctx render = do
  save ctx
  rotate (Math.tau / 3.0) ctx
  render
  restore ctx
```

In the interest of abstracting over common use cases using higher-order functions, the `canvas` library provides the `withContext` function, which performs some canvas action while preserving the original context state:

```
withContext
  :: Context2D
  -> Effect a
  -> Effect a
```

We could rewrite the `rotated` function above using `withContext` as follows:

```
rotated ctx render =
  withContext ctx do
    rotate (Math.tau / 3.0) ctx
    render
```

Global Mutable State

In this section, we'll use the `refs` package to demonstrate another effect in the `Effect` monad.

The `Effect.Ref` module provides a type constructor for global mutable references and an associated effect:

```
> import Effect.Ref

> :kind Ref
Type -> Type
```

A value of type `Ref a` is a mutable reference cell containing a value of type `a`, used to track global mutation. As such, it should be used sparingly.

The `Example/Refs.purs` file contains an example that uses a `Ref` to track mouse clicks on the `canvas` element.

The code starts by creating a new reference containing the value `0` by using the `new` action:

```
  clickCount <- Ref.new 0
```

Inside the click event handler, the `modify` action is used to update the click count, and the updated value is returned.

```
    count <- Ref.modify (\count -> count + 1) clickCount
```

In the `render` function, the click count is used to determine the transformation applied to a rectangle:

```
  withContext ctx do
    let scaleX = Number.sin (toNumber count * Number.tau / 8.0) + 1.5
    let scaleY = Number.sin (toNumber count * Number.tau / 12.0) + 1.5

    translate ctx { translateX: 300.0, translateY:  300.0 }
    rotate ctx (toNumber count * Number.tau / 36.0)
    scale ctx { scaleX: scaleX, scaleY: scaleY }
    translate ctx { translateX: -100.0, translateY: -100.0 }

    fillPath ctx $ rect ctx
      { x: 0.0
      , y: 0.0
      , width: 200.0
      , height: 200.0
      }
```

This action uses `withContext` to preserve the original transformation and then applies the following sequence of transformations (remember that transformations are applied bottom-to-top):

- The rectangle is translated through `(-100, -100)`, so its center lies at the origin.
- The rectangle is scaled around the origin.
- The rectangle is rotated through some multiple of `10` degrees around the origin.
- The rectangle is translated through `(300, 300)`, so its center lies at the center of the canvas.

Build the example:

```
$ spago bundle-app --main Example.Refs --to dist/Main.js
```

and open the `html/index.html` file. If you click the canvas repeatedly, you should see a green rectangle rotating around the center of the canvas.

Exercises

1. (Easy) Write a higher-order function that simultaneously strokes and fills a path. Rewrite the `Random.purs` example using your function.
2. (Medium) Use `Random` and `Dom` to create an application that renders a circle with random position, color, and radius to the canvas when the mouse is clicked.
3. (Medium) Write a function that transforms the scene by rotating it around a point with specified coordinates. *Hint*: use a translation to first translate the scene to the origin.

L-Systems

In this final example, we will use the `canvas` package to write a function for rendering *L-systems* (or *Lindenmayer systems*).

An L-system is defined by an *alphabet*, an initial sequence of letters from the alphabet, and a set of *production rules*. Each production rule takes a letter of the alphabet and returns a sequence of replacement letters. This process is iterated some number of times, starting with the initial sequence of letters.

If each letter of the alphabet is associated with some instruction to perform on the canvas, the L-system can be rendered by following the instructions in order.

For example, suppose the alphabet consists of the letters `L` (turn left), `R` (turn right), and `F` (move forward). We might define the following production rules:

```
L -> L
R -> R
F -> FLFRRFLF
```

If we start with the initial sequence "FRRFRRFRR" and iterate, we obtain the following sequence:

```
FRRFRRFRR
FLFRRFLFRRFLFRRFLFRRFLFRRFLFRR
FLFRRFLFLFLFRRFLFRRFLFRRFLFLFLFRRFLFRRFLFRRFLF...
```

and so on. Plotting a piecewise-linear path corresponding to this set of instructions approximates the *Koch curve*. Increasing the number of iterations increases the resolution of the curve.

Let's translate this into the language of types and functions.

We can represent our alphabet of letters with the following ADT:

```
data Letter = L | R | F
```

This data type defines one data constructor for each letter in our alphabet.

How can we represent the initial sequence of letters? Well, that's just an array of letters from our alphabet, which we will call a `Sentence`:

```
type Sentence = Array Letter

initial :: Sentence
initial = [F, R, R, F, R, R, F, R, R]
```

Our production rules can be represented as a function from `Letter` to `Sentence` as follows:

```
productions :: Letter -> Sentence
productions L = [L]
productions R = [R]
productions F = [F, L, F, R, R, F, L, F]
```

This is just copied straight from the specification above.

Now we can implement a function `lsystem` that will take a specification in this form and render it to the canvas. What type should `lsystem` have? Well, it needs to take values like `initial` and `productions` as arguments, as well as a function that can render a letter of the alphabet to the canvas.

Here is a first approximation to the type of `lsystem`:

```
Sentence
-> (Letter -> Sentence)
-> (Letter -> Effect Unit)
-> Int
-> Effect Unit
```

The first two argument types correspond to the values `initial` and `productions`.

The third argument represents a function that takes a letter of the alphabet and *interprets* it by performing some actions on the canvas. In our example, this would mean turning left in the case of the letter `L`, turning right in the case of the letter `R`, and moving forward in the case of a letter `F`.

The final argument is a number representing the number of iterations of the production rules we would like to perform.

The first observation is that the `lsystem` function should work for only one type of `Letter`, but for any type, so we should generalize our type accordingly. Let's replace `Letter` and `Sentence` with `a` and `Array a` for some quantified type variable `a`:

```
forall a. Array a
          -> (a -> Array a)
          -> (a -> Effect Unit)
          -> Int
          -> Effect Unit
```

The second observation is that, to implement instructions like "turn left" and "turn right", we will need to maintain some state, namely the direction in which the path is moving at any time. We need to modify our function to pass the state through the computation. Again, the `lsystem` function should work for any type of state, so we will represent it using the type variable `s`.

We need to add the type `s` in three places:

```
forall a s. Array a
            -> (a -> Array a)
            -> (s -> a -> Effect s)
            -> Int
            -> s
            -> Effect s
```

Firstly, the type `s` was added as the type of an additional argument to `lsystem`. This argument will represent the initial state of the L-system.

The type `s` also appears as an argument to, and as the return type of the interpretation function (the third argument to `lsystem`). The interpretation function will now receive the current state of the L-system as an argument, and will return a new, updated state as its return value.

In the case of our example, we can define use following type to represent the state:

```
type State =
  { x :: Number
  , y :: Number
  , theta :: Number
  }
```

The properties `x` and `y` represent the current position of the path, and the `theta` property represents the current direction of the path, specified as the angle between the path direction and the horizontal axis, in radians.

The initial state of the system might be specified as follows:

```
initialState :: State
initialState = { x: 120.0, y: 200.0, theta: 0.0 }
```

Now let's try to implement the `lsystem` function. We will find that its definition is remarkably simple.

It seems reasonable that `lsystem` should recurse on its fourth argument (of type `Int`). On each step of the recursion, the current sentence will change, having been updated by using the production rules. With that in mind, let's begin by introducing names for the function arguments, and delegating to a helper function:

```
lsystem :: forall a s
         . Array a
        -> (a -> Array a)
        -> (s -> a -> Effect s)
        -> Int
        -> s
        -> Effect s
lsystem init prod interpret n state = go init n
  where
```

The `go` function works by recursion on its second argument. There are two cases: when `n` is zero and `n` is non-zero.

In the first case, the recursion is complete, and we need to interpret the current sentence according to the interpretation function. We have a sentence of type `Array a`, a state of type `s`, and a function of type `s -> a -> Effect s`. This sounds like a job for the `foldM` function which we defined earlier, and which is available from the `control` package:

```
  go s 0 = foldM interpret state s
```

What about in the non-zero case? In that case, we can simply apply the production rules to each letter of the current sentence, concatenate the results, and repeat by calling `go` recursively:

```
  go s i = go (concatMap prod s) (i - 1)
```

That's it! Note how using higher-order functions like `foldM` and `concatMap` allowed us to communicate our ideas concisely.

However, we're not quite done. The type we have given is actually still too specific. Note that we don't use any canvas operations anywhere in our implementation. Nor do we make use of the structure of the `Effect` monad at all. In fact, our function works for *any* monad `m`!

Here is the more general type of `lsystem`, as specified in the accompanying source code for this chapter:

```
lsystem :: forall a m s
         . Monad m
        => Array a
        -> (a -> Array a)
        -> (s -> a -> m s)
        -> Int
        -> s
        -> m s
```

We can understand this type as saying that our interpretation function is free to have any side-effects at all, captured by the monad `m`. It might render to the canvas, print information to the console, or support failure or multiple return values. The reader is encouraged to try writing L-systems that use these various types of side-effect.

This function is a good example of the power of separating data from implementation. The advantage of this approach is that we can interpret our data in multiple ways. We might even factor `lsystem` into two smaller functions: the first would build the sentence using repeated application of `concatMap`, and the second would interpret the sentence using `foldM`. This is also left as an exercise for the reader.

Let's complete our example by implementing its interpretation function. The type of `lsystem` tells us that its type signature must be `s -> a -> m s` for some types `a` and `s` and a type constructor `m`. We know that we want `a` to be `Letter` and `s` to be `State`, and for the monad `m` we can choose `Effect`. This gives us the following type:

```
interpret :: State -> Letter -> Effect State
```

To implement this function, we need to handle the three data constructors of the `Letter` type. To interpret the letters `L` (move left) and `R` (move right), we simply have to update the state to change the angle `theta` appropriately:

```
interpret state L = pure $ state { theta = state.theta - Number.tau / 6.0 }
interpret state R = pure $ state { theta = state.theta + Number.tau / 6.0 }
```

To interpret the letter `F` (move forward), we can calculate the new position of the path, render a line segment, and update the state as follows:

```
interpret state F = do
  let x = state.x + Number.cos state.theta * 1.5
      y = state.y + Number.sin state.theta * 1.5
  moveTo ctx state.x state.y
  lineTo ctx x y
  pure { x, y, theta: state.theta }
```

Note that in the source code for this chapter, the `interpret` function is defined using a `let` binding inside the `main` function, so that the name `ctx` is in scope. It would also be possible to move the context into the `State` type, but this would be inappropriate because it is not a changing part of the state of the system.

To render this L-system, we can simply use the `strokePath` action:

```
strokePath ctx $ lsystem initial productions interpret 5 initialState
```

Compile the L-system example using

```
$ spago bundle-app --main Example.LSystem --to dist/Main.js
```

and open `html/index.html`. You should see the Koch curve rendered to the canvas.

Exercises

1. (Easy) Modify the L-system example above to use `fillPath` instead of `strokePath`. *Hint*: you will need to include a call to `closePath`, and move the call to `moveTo` outside of the `interpret` function.

2. (Easy) Try changing the various numerical constants in the code to understand their effect on the rendered system.

3. (Medium) Break the `lsystem` function into two smaller functions. The first should build the final sentence using repeated application of `concatMap`, and the second should use `foldM` to interpret the result.

4. (Medium) Add a drop shadow to the filled shape using the `setShadowOffsetX`, `setShadowOffsetY`, `setShadowBlur`, and `setShadowColor` actions. *Hint*: use PSCi to find the types of these functions.

5. (Medium) The angle of the corners is currently a constant $\tau/6$. Instead, it can be moved into the `Letter` data type, which allows it to be changed by the production

rules:

```
type Angle = Number

data Letter = L Angle | R Angle | F
```

How can this new information be used in the production rules to create interesting shapes?

6. (Difficult) An L-system is given by an alphabet with four letters: `L` (turn left through 60 degrees), `R` (turn right through 60 degrees), `F` (move forward), and `M` (also move forward).

   The initial sentence of the system is the single letter `M`.

   The production rules are specified as follows:

   ```
   L -> L
   R -> R
   F -> FLMLFRMRFRMRFLMLF
   M -> MRFRMLFLMLFLMRFRM
   ```

   Render this L-system. *Note*: you will need to decrease the number of iterations of the production rules since the size of the final sentence grows exponentially with the number of iterations.

   Now, notice the symmetry between `L` and `M` in the production rules. The two "move forward" instructions can be differentiated using a `Boolean` value using the following alphabet type:

   ```
   data Letter = L | R | F Boolean
   ```

   Implement this L-system again using this representation of the alphabet.

7. (Difficult) Use a different monad `m` in the interpretation function. You might try using `Effect.Console` to write the L-system onto the console, or using `Effect.Random` to apply random "mutations" to the state type.

Conclusion

In this chapter, we learned how to use the HTML5 Canvas API from PureScript by using the `canvas` library. We also saw a practical demonstration of many techniques we have learned already: maps and folds, records and row polymorphism, and the `Effect` monad for handling side-effects.

The examples also demonstrated the power of higher-order functions and *separating data from implementation*. It would be possible to extend these ideas to completely separate the representation of a scene from its rendering function, using an algebraic data type, for example:

```
data Scene
  = Rect Rectangle
  | Arc Arc
  | PiecewiseLinear (Array Point)
  | Transformed Transform Scene
  | Clipped Rectangle Scene
  | ...
```

This approach is taken in the `drawing` package, and it brings the flexibility of manipulating the scene as data in various ways before rendering.

For examples of games rendered to the canvas, see the "Behavior" and "Signal" recipes in the cookbook.

Generative Testing

Chapter Goals

In this chapter, we will see a particularly elegant application of type classes to the problem of testing. Instead of testing our code by telling the compiler *how* to test, we simply assert *what* properties our code should have. Test cases can be generated randomly from this specification, using type classes to hide the boilerplate code of random data generation. This is called *generative testing* (or *property-based testing*), a technique made popular by the QuickCheck library in Haskell.

The `quickcheck` package is a port of Haskell's QuickCheck library to PureScript, and for the most part, it preserves the types and syntax of the original library. We will see how to use `quickcheck` to test a simple library, using Spago to integrate our test suite into our development process.

Project Setup

This chapter's project adds `quickcheck` as a dependency.

In a Spago project, test sources should be placed in the `test` directory, and the main module for the test suite should be named `Test.Main`. The test suite can be run using the `spago test` command.

Writing Properties

The `Merge` module implements a simple function `merge`, which we will use to demonstrate the features of the `quickcheck` library.

```
merge :: Array Int -> Array Int -> Array Int
```

`merge` takes two sorted arrays of integers and merges their elements so that the result is also sorted. For example:

```
> import Merge
> merge [1, 3, 5] [2, 4, 5]

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

In a typical test suite, we might test `merge` by generating a few small test cases like this by hand and asserting that the results were equal to the appropriate values. However, everything we need to know about the `merge` function can be summarized by this property:

- If `xs` and `ys` are sorted, then `merge xs ys` is the sorted result of both arrays appended together.

`quickcheck` allows us to test this property directly by generating random test cases. We state the properties we want our code to have as functions. In this case, we have a single property:

```
main = do
  quickCheck \xs ys ->
    eq (merge (sort xs) (sort ys)) (sort $ xs <> ys)
```

When we run this code, `quickcheck` will attempt to disprove the properties we claimed by generating random inputs `xs` and `ys` and passing them to our functions. If our function returns `false` for any inputs, the property will be incorrect, and the library will raise an error. Fortunately, the library is unable to disprove our properties after generating 100 random test cases:

```
$ spago test

Installation complete.
Build succeeded.
100/100 test(s) passed.
...
Tests succeeded.
```

If we deliberately introduce a bug into the `merge` function (for example, by changing the less-than check for a greater-than check), then an exception is thrown at runtime after the first failed test case:

```
Error: Test 1 failed:
Test returned false
```

As we can see, this error message is not very helpful, but it can be improved with a little work.

Improving Error Messages

To provide error messages along with our failed test cases, `quickcheck` provides the `<?>` operator. Simply separate the property definition from the error message using `<?>`, as follows:

```
quickCheck \xs ys ->
  let
    result = merge (sort xs) (sort ys)
    expected = sort $ xs <> ys
  in
    eq result expected <?> "Result:\n" <> show result <> "\nnot equal to
expected:\n" <> show expected
```

This time, if we modify the code to introduce a bug, we see our improved error message after the first failed test case:

```
Error: Test 1 (seed 534161891) failed:
Result:
[-822215,-196136,-116841,618343,887447,-888285]
not equal to expected:
[-888285,-822215,-196136,-116841,618343,887447]
```

Notice how the input `xs` and `ys` were generated as arrays of randomly-selected integers.

Exercises

1. (Easy) Write a property that asserts that merging an array with an empty one does not modify the original array. *Note*: This new property is redundant since this situation is already covered by our existing property. We're just trying to give readers a simple way to practice using quickCheck.
2. (Easy) Add an appropriate error message to the remaining property for `merge`.

Testing Polymorphic Code

The `Merge` module defines a generalization of the `merge` function, called `mergePoly`, which works not only with arrays of numbers, but also arrays of any type belonging to the `Ord` type class:

```
mergePoly :: forall a. Ord a => Array a -> Array a -> Array a
```

If we modify our original test to use `mergePoly` in place of `merge`, we see the following error message:

```
No type class instance was found for

  Test.QuickCheck.Arbitrary.Arbitrary t0

The instance head contains unknown type variables.
Consider adding a type annotation.
```

This error message indicates that the compiler could not generate random test cases because it did not know what type of elements we wanted our arrays to have. In these sorts of cases, we can use type annotations to force the compiler to infer a particular type, such as `Array Int`:

```
quickCheck \xs ys ->
  eq (mergePoly (sort xs) (sort ys) :: Array Int) (sort $ xs <> ys)
```

We can alternatively use a helper function to specify the type, which may result in cleaner code. For example, if we define a function `ints` as a synonym for the identity function:

```
ints :: Array Int -> Array Int
ints = id
```

then we can modify our test so that the compiler infers the type `Array Int` for our two array arguments:

```
quickCheck \xs ys ->
  eq (ints $ mergePoly (sort xs) (sort ys)) (sort $ xs <> ys)
```

Here, `xs` and `ys` have type `Array Int` since the `ints` function has been used to disambiguate the unknown type.

Exercises

1. (Easy) Write a function `bools` that forces the types of `xs` and `ys` to be `Array Boolean`, and add additional properties that test `mergePoly` at that type.
2. (Medium) Choose a pure function from the core libraries (for example, from the

`arrays` package), and write a QuickCheck property for it, including an appropriate error message. Your property should use a helper function to fix any polymorphic type arguments to either `Int` or `Boolean`.

Generating Arbitrary Data

Now we will see how the `quickcheck` library can randomly generate test cases for our properties.

Those types whose values can be randomly generated are captured by the `Arbitrary` type class:

```
class Arbitrary t where
  arbitrary :: Gen t
```

The `Gen` type constructor represents the side-effects of *deterministic random data generation*. It uses a pseudo-random number generator to generate deterministic random function arguments from a seed value. The `Test.QuickCheck.Gen` module defines several useful combinators for building generators.

`Gen` is also a monad and an applicative functor, so we have the usual collection of combinators at our disposal for creating new instances of the `Arbitrary` type class.

For example, we can use the `Arbitrary` instance for the `Int` type, provided in the `quickcheck` library, to create a distribution on the 256-byte values, using the `Functor` instance for `Gen` to map a function from integers to bytes over arbitrary integer values:

```
newtype Byte = Byte Int

instance Arbitrary Byte where
  arbitrary = map intToByte arbitrary
    where
    intToByte n | n >= 0 = Byte (n `mod` 256)
                | otherwise = intToByte (-n)
```

Here, we define a type `Byte` of integral values between 0 and 255. The `Arbitrary` instance uses the `map` function to lift the `intToByte` function over the `arbitrary` action. The type of the inner `arbitrary` action is inferred as `Gen Int`.

We can also use this idea to improve our test for `merge`:

```
quickCheck \xs ys ->
  eq (numbers $ mergePoly (sort xs) (sort ys)) (sort $ xs <> ys)
```

In this test, we generated arbitrary arrays `xs` and `ys`, but had to sort them, since `merge` expects sorted input. On the other hand, we could create a newtype representing sorted arrays and write an `Arbitrary` instance that generates sorted data:

```
newtype Sorted a = Sorted (Array a)

sorted :: forall a. Sorted a -> Array a
sorted (Sorted xs) = xs

instance (Arbitrary a, Ord a) => Arbitrary (Sorted a) where
  arbitrary = map (Sorted <<< sort) arbitrary
```

With this type constructor, we can modify our test as follows:

```
quickCheck \xs ys ->
  eq (ints $ mergePoly (sorted xs) (sorted ys)) (sort $ sorted xs <> sorted ys)
```

This may look like a small change, but the types of `xs` and `ys` have changed to `Sorted Int` instead of just `Array Int`. This communicates our *intent* in a clearer way – the `mergePoly` function takes sorted input. Ideally, the type of the `mergePoly` function itself would be updated to use the `Sorted` type constructor.

As a more interesting example, the `Tree` module defines a type of sorted binary trees with values at the branches:

```
data Tree a
  = Leaf
  | Branch (Tree a) a (Tree a)
```

The `Tree` module defines the following API:

```
insert    :: forall a. Ord a => a -> Tree a -> Tree a
member    :: forall a. Ord a => a -> Tree a -> Boolean
fromArray :: forall a. Ord a => Array a -> Tree a
toArray   :: forall a. Tree a -> Array a
```

The `insert` function inserts a new element into a sorted tree, and the `member` function can query a tree for a particular value. For example:

```
> import Tree

> member 2 $ insert 1 $ insert 2 Leaf
true

> member 1 Leaf
false
```

The `toArray` and `fromArray` functions can convert sorted trees to and from arrays. We can use `fromArray` to write an `Arbitrary` instance for trees:

```
instance (Arbitrary a, Ord a) => Arbitrary (Tree a) where
  arbitrary = map fromArray arbitrary
```

We can now use `Tree a` as the type of an argument to our test properties whenever there is an `Arbitrary` instance available for the type `a`. For example, we can test that the `member` test always returns `true` after inserting a value:

```
quickCheck \t a ->
  member a $ insert a $ treeOfInt t
```

Here, the argument `t` is a randomly-generated tree of type `Tree Int`, where the type argument disambiguated by the identity function `treeOfInt`.

Exercises

1. (Medium) Create a newtype for `String` with an associated `Arbitrary` instance which generates collections of randomly-selected characters in the range `a-z`. *Hint*: use the `elements` and `arrayOf` functions from the `Test.QuickCheck.Gen` module.
2. (Difficult) Write a property that asserts that a value inserted into a tree is still a member of that tree after arbitrarily many more insertions.

Testing Higher-Order Functions

The `Merge` module defines another generalization of the `merge` function – the `mergeWith` function takes an additional function as an argument to determine the order in which elements should be merged. That is, `mergeWith` is a higher-order function.

For example, we can pass the `length` function as the first argument to merge two arrays already in length-increasing order. The result should also be in length-increasing order:

```
> import Data.String

> mergeWith length
    ["", "ab", "abcd"]
    ["x", "xyz"]

["","x","ab","xyz","abcd"]
```

How might we test such a function? Ideally, we would like to generate values for all three arguments, including the first argument, which is a function.

There is a second type class that allows us to create randomly-generated functions. It is called `Coarbitrary`, and it is defined as follows:

```
class Coarbitrary t where
  coarbitrary :: forall r. t -> Gen r -> Gen r
```

The `coarbitrary` function takes a function argument of type `t` and a random generator for a function result of type `r`. It uses the function argument to *perturb* the random generator. That is, it uses the function argument to modify the random output of the random generator for the result.

In addition, there is a type class instance that gives us `Arbitrary` functions if the function domain is `Coarbitrary` and the function codomain is `Arbitrary`:

```
instance (Coarbitrary a, Arbitrary b) => Arbitrary (a -> b)
```

In practice, we can write properties that take functions as arguments. In the case of the `mergeWith` function, we can generate the first argument randomly, modifying our tests to take account of the new argument.

We cannot guarantee that the result will be sorted – we do not even necessarily have an `Ord` instance – but we can expect that the result be sorted with respect to the function `f` that we pass in as an argument. In addition, we need the two input arrays to be sorted concerning `f`, so we use the `sortBy` function to sort `xs` and `ys` based on comparison after the function `f` has been applied:

```
quickCheck \xs ys f ->
  let
    result =
      map f $
        mergeWith (intToBool f)
                  (sortBy (compare `on` f) xs)
                  (sortBy (compare `on` f) ys)
    expected =
      map f $
        sortBy (compare `on` f) $ xs <> ys
  in
    eq result expected
```

Here, we use a function `intToBool` to disambiguate the type of the function `f`:

```
intToBool :: (Int -> Boolean) -> Int -> Boolean
intToBool = id
```

In addition to being `Arbitrary`, functions are also `Coarbitrary`:

```
instance (Arbitrary a, Coarbitrary b) => Coarbitrary (a -> b)
```

This means that we are not limited to just values and functions – we can also randomly generate *higher-order functions*, or functions whose arguments are higher-order functions, and so on.

Writing Coarbitrary Instances

Just as we can write `Arbitrary` instances for our data types by using the `Monad` and `Applicative` instances of `Gen`, we can write our own `Coarbitrary` instances as well. This allows us to use our own data types as the domain of randomly-generated functions.

Let's write a `Coarbitrary` instance for our `Tree` type. We will need a `Coarbitrary` instance for the type of the elements stored in the branches:

```
instance Coarbitrary a => Coarbitrary (Tree a) where
```

We have to write a function that perturbs a random generator given a value of type `Tree a`. If the input value is a `Leaf`, then we will return the generator unchanged:

```
  coarbitrary Leaf = id
```

If the tree is a `Branch`, then we will perturb the generator using the left subtree, the value, and the right subtree. We use function composition to create our perturbing function:

```
  coarbitrary (Branch l a r) =
    coarbitrary l <<<
    coarbitrary a <<<
    coarbitrary r
```

Now we can write properties whose arguments include functions taking trees as arguments. For example, the `Tree` module defines a function `anywhere`, which tests if a predicate holds on any subtree of its argument:

```
anywhere :: forall a. (Tree a -> Boolean) -> Tree a -> Boolean
```

Now we can generate the predicate function randomly. For example, we expect the `anywhere` function to *respect disjunction*:

```
quickCheck \f g t ->
  anywhere (\s -> f s || g s) t ==
    anywhere f (treeOfInt t) || anywhere g t
```

Here, the `treeOfInt` function is used to fix the type of values contained in the tree to the type `Int`:

```
treeOfInt :: Tree Int -> Tree Int
treeOfInt = id
```

Testing Without Side-Effects

For the purposes of testing, we usually include calls to the `quickCheck` function in the `main` action of our test suite. However, there is a variant of the `quickCheck` function, called `quickCheckPure` which does not use side-effects. Instead, it is a pure function that takes a random seed as an input and returns an array of test results.

We can test `quickCheckPure` using PSCi. Here, we test that the `merge` operation is associative:

```
> import Prelude
> import Merge
> import Test.QuickCheck
> import Test.QuickCheck.LCG (mkSeed)

> :paste
… quickCheckPure (mkSeed 12345) 10 \xs ys zs ->
…   ((xs `merge` ys) `merge` zs) ==
…     (xs `merge` (ys `merge` zs))
… ^D

Success : Success : ...
```

`quickCheckPure` takes three arguments: the random seed, the number of test cases to generate, and the property to test. If all tests pass, you should see an array of `Success` data constructors printed to the console.

`quickCheckPure` might be useful in other situations, such as generating random input data for performance benchmarks or sample form data for web applications.

Exercises

1. (Easy) Write `Coarbitrary` instances for the `Byte` and `Sorted` type constructors.
2. (Medium) Write a (higher-order) property which asserts associativity of the `mergeWith f` function for any function `f`. Test your property in PSCi using `quickCheckPure`.
3. (Medium) Write `Arbitrary` and `Coarbitrary` instances for the following data type:

   ```
   data OneTwoThree a = One a | Two a a | Three a a a
   ```

   Hint: Use the `oneOf` function defined in `Test.QuickCheck.Gen` to define your `Arbitrary` instance.
4. (Medium) Use `all` to simplify the result of the `quickCheckPure` function – your new function should have the type `List Result -> Boolean` and should return `true` if every test passes and `false` otherwise.
5. (Medium) As another approach to simplifying the result of `quickCheckPure`, try writing a function `squashResults :: List Result -> Result`. Consider using the `First` monoid from `Data.Maybe.First` with the `foldMap` function to preserve the first error in case of failure.

Conclusion

In this chapter, we met the `quickcheck` package, which can be used to write tests in a declarative way using the paradigm of *generative testing*. In particular:

- We saw how to automate QuickCheck tests using `spago test`.
- We saw how to write properties as functions and how to use the `<?>` operator to improve error messages.
- We saw how the `Arbitrary` and `Coarbitrary` type classes enable generation of boilerplate testing code and how they allow us to test higher-order properties.
- We saw how to implement custom `Arbitrary` and `Coarbitrary` instances for our own data types.

Domain-Specific Languages

Chapter Goals

In this chapter, we will explore the implementation of *domain-specific languages* (or *DSLs*) in PureScript, using a number of standard techniques.

A domain-specific language is a language that is well-suited to development in a particular problem domain. Its syntax and functions are chosen to maximize the readability of code used to express ideas in that domain. We have already seen several examples of domain-specific languages in this book:

- The `Game` monad and its associated actions, developed in chapter 11, constitute a domain-specific language for the domain of *text adventure game development*.
- The `quickcheck` package, covered in Chapter 13, is a domain-specific language for the domain of *generative testing*. Its combinators enable a particularly expressive notation for test properties.

This chapter will take a more structured approach to some standard techniques in implementing domain-specific languages. It is by no means a complete exposition of the subject, but should provide you with enough knowledge to build some practical DSLs for your own tasks.

Our running example will be a domain-specific language for creating HTML documents. We will aim to develop a type-safe language for describing correct HTML documents, and we will work by improving a naive implementation in small steps.

Project Setup

The project accompanying this chapter adds one new dependency – the `free` library, which defines the *free monad*, one of the tools we will use.

We will test this chapter's project in PSCi.

An HTML Data Type

The most basic version of our HTML library is defined in the `Data.DOM.Simple` module. The module contains the following type definitions:

```
newtype Element = Element
  { name          :: String
  , attribs       :: Array Attribute
  , content       :: Maybe (Array Content)
  }

data Content
  = TextContent String
  | ElementContent Element

newtype Attribute = Attribute
  { key           :: String
  , value         :: String
  }
```

The `Element` type represents HTML elements. Each element consists of an element name, an array of attribute pairs, and some content. The content property uses the `Maybe` type to indicate that an element might be open (containing other elements and text) or closed.

The key function of our library is a function

```
render :: Element -> String
```

which renders HTML elements as HTML strings. We can try out this version of the library by constructing values of the appropriate types explicitly in PSCi:

```
$ spago repl

> import Prelude
> import Data.DOM.Simple
> import Data.Maybe
> import Effect.Console

> :paste
… log $ render $ Element
…   { name: "p"
…   , attribs: [
…       Attribute
…         { key: "class"
…         , value: "main"
…         }
…     ]
…   , content: Just [
…       TextContent "Hello World!"
…     ]
…   }
… ^D

<p class="main">Hello World!</p>
unit
```

As it stands, there are several problems with this library:

- Creating HTML documents is difficult – every new element requires at least one record and one data constructor.
- It is possible to represent invalid documents:
  - The developer might mistype the element name
  - The developer can associate an attribute with the wrong type of element
  - The developer can use a closed element when an open element is correct

In the remainder of the chapter, we will apply certain techniques to solve these problems and turn our library into a usable domain-specific language for creating HTML documents.

Smart Constructors

The first technique we will apply is simple but can be very effective. Instead of exposing the representation of the data to the module's users, we can use the module exports list to hide the `Element`, `Content`, and `Attribute` data constructors, and only export so-called *smart constructors*, which construct data known to be correct.

Here is an example. First, we provide a convenience function for creating HTML elements:

```
element :: String -> Array Attribute -> Maybe (Array Content) -> Element
element name attribs content = Element
  { name:      name
  , attribs:   attribs
  , content:   content
  }
```

Next, we create smart constructors for those HTML elements we want our users to be able to create by applying the `element` function:

```
a :: Array Attribute -> Array Content -> Element
a attribs content = element "a" attribs (Just content)

p :: Array Attribute -> Array Content -> Element
p attribs content = element "p" attribs (Just content)

img :: Array Attribute -> Element
img attribs = element "img" attribs Nothing
```

Finally, we update the module exports list to only export those functions which are known to construct correct data structures:

```
module Data.DOM.Smart
  ( Element
  , Attribute(..)
  , Content(..)

  , a
  , p
  , img

  , render
  ) where
```

The module exports list is provided immediately after the module name inside parentheses. Each module export can be one of three types:

- A value (or function), indicated by the name of the value,
- A type class, indicated by the name of the class,
- A type constructor and any associated data constructors indicated by the name of the type followed by a parenthesized list of exported data constructors.

Here, we export the `Element` *type*, but we do not export its data constructors. If we did, the user could construct invalid HTML elements.

In the case of the `Attribute` and `Content` types, we still export all of the data constructors

(indicated by the symbol `..` in the exports list). We will apply the technique of smart constructors to these types shortly.

Notice that we have already made some big improvements to our library:

- It is impossible to represent HTML elements with invalid names (of course, we are restricted to the set of element names provided by the library).
- Closed elements cannot contain content by construction.

We can apply this technique to the `Content` type very easily. We simply remove the data constructors for the `Content` type from the exports list and provide the following smart constructors:

```
text :: String -> Content
text = TextContent

elem :: Element -> Content
elem = ElementContent
```

Let's apply the same technique to the `Attribute` type. First, we provide a general-purpose smart constructor for attributes. Here is a first attempt:

```
attribute :: String -> String -> Attribute
attribute key value = Attribute
  { key: key
  , value: value
  }

infix 4 attribute as :=
```

This representation suffers from the same problem as the original `Element` type – it is possible to represent attributes that do not exist or whose names were entered incorrectly. To solve this problem, we can create a newtype that represents attribute names:

```
newtype AttributeKey = AttributeKey String
```

With that, we can modify our operator as follows:

```
attribute :: AttributeKey -> String -> Attribute
attribute (AttributeKey key) value = Attribute
  { key: key
  , value: value
  }
```

If we do not export the `AttributeKey` data constructor, then the user has no way to

construct values of type `AttributeKey` other than by using functions we explicitly export. Here are some examples:

```
href :: AttributeKey
href = AttributeKey "href"

_class :: AttributeKey
_class = AttributeKey "class"

src :: AttributeKey
src = AttributeKey "src"

width :: AttributeKey
width = AttributeKey "width"

height :: AttributeKey
height = AttributeKey "height"
```

Here is the final exports list for our new module. Note that we no longer export any data constructors directly:

```
module Data.DOM.Smart
  ( Element
  , Attribute
  , Content
  , AttributeKey

  , a
  , p
  , img

  , href
  , _class
  , src
  , width
  , height

  , attribute, (:=)
  , text
  , elem

  , render
  ) where
```

If we try this new module in PSCi, we can already see massive improvements in the conciseness of the user code:

```
$ spago repl

> import Prelude
> import Data.DOM.Smart
> import Effect.Console
> log $ render $ p [ _class := "main" ] [ text "Hello World!" ]

<p class="main">Hello World!</p>
unit
```

Note, however, that no changes had to be made to the `render` function, because the underlying data representation never changed. This is one of the benefits of the smart constructors approach – it allows us to separate the internal data representation for a module from the representation perceived by users of its external API.

Exercises

1. (Easy) Use the `Data.DOM.Smart` module to experiment by creating new HTML documents using `render`.

2. (Medium) Some HTML attributes, such as `checked` and `disabled`, do not require values and may be rendered as *empty attributes*:

   ```
   <input disabled>
   ```

   Modify the representation of an `Attribute` to take empty attributes into account. Write a function which can be used in place of `attribute` or `:=` to add an empty attribute to an element.

Phantom Types

To motivate the next technique, consider the following code:

```
> log $ render $ img
    [ src    := "cat.jpg"
    , width  := "foo"
    , height := "bar"
    ]

<img src="cat.jpg" width="foo" height="bar" />
unit
```

The problem here is that we have provided string values for the `width` and `height` attributes, where we should only be allowed to provide numeric values in units of pixels or percentage points.

To solve this problem, we can introduce a so-called *phantom type* argument to our `AttributeKey` type:

```
newtype AttributeKey a = AttributeKey String
```

The type variable `a` is called a *phantom type* because there are no values of type `a` involved in the right-hand side of the definition. The type `a` only exists to provide more information at compile-time. Any value of type `AttributeKey a` is simply a string at runtime, but at compile-time, the type of the value tells us the desired type of the values associated with this key.

We can modify the type of our `attribute` function to take the new form of `AttributeKey` into account:

```
attribute :: forall a. IsValue a => AttributeKey a -> a -> Attribute
attribute (AttributeKey key) value = Attribute
  { key: key
  , value: toValue value
  }
```

Here, the phantom type argument `a` is used to ensure that the attribute key and attribute value have compatible types. Since the user cannot create values of type `AttributeKey a` directly (only via the constants we provide in the library), every attribute will be correct by construction.

Note that the `IsValue` constraint ensures that whatever value type we associate to a key, its values can be converted to strings and displayed in the generated HTML. The `IsValue` type class is defined as follows:

```
class IsValue a where
  toValue :: a -> String
```

We also provide type class instances for the `String` and `Int` types:

```
instance IsValue String where
  toValue = id

instance IsValue Int where
  toValue = show
```

We also have to update our `AttributeKey` constants so that their types reflect the new type parameter:

```
href :: AttributeKey String
href = AttributeKey "href"

_class :: AttributeKey String
_class = AttributeKey "class"

src :: AttributeKey String
src = AttributeKey "src"

width :: AttributeKey Int
width = AttributeKey "width"

height :: AttributeKey Int
height = AttributeKey "height"
```

Now we find it is impossible to represent these invalid HTML documents, and we are forced to use numbers to represent the `width` and `height` attributes instead:

```
> import Prelude
> import Data.DOM.Phantom
> import Effect.Console

> :paste
… log $ render $ img
…   [ src    := "cat.jpg"
…   , width  := 100
…   , height := 200
…   ]
… ^D

<img src="cat.jpg" width="100" height="200" />
unit
```

Exercises

1. (Easy) Create a data type representing either pixel or percentage lengths. Write an instance of `IsValue` for your type. Modify the `width` and `height` attributes to use your new type.

2. (Difficult) By defining type-level representatives for the Boolean values `true` and `false`, we can use a phantom type to encode whether an `AttributeKey` represents an *empty attribute*, such as `disabled` or `checked`.

   ```
   data True
   data False
   ```

   Modify your solution to the previous exercise to use a phantom type to prevent the user from using the `attribute` operator with an empty attribute.

The Free Monad

In our final set of modifications to our API, we will use a construction called the *free monad* to turn our `Content` type into a monad, enabling do notation. This will allow us to structure our HTML documents in a form in which the nesting of elements becomes clearer – instead of this:

```
p [ _class := "main" ]
  [ elem $ img
      [ src    := "cat.jpg"
      , width  := 100
      , height := 200
      ]
  , text "A cat"
  ]
```

we will be able to write this:

```
p [ _class := "main" ] $ do
  elem $ img
    [ src    := "cat.jpg"
    , width  := 100
    , height := 200
    ]
  text "A cat"
```

However, do notation is not the only benefit of a free monad. The free monad allows us to separate the *representation* of our monadic actions from their *interpretation* and even support *multiple interpretations* of the same actions.

The `Free` monad is defined in the `free` library in the `Control.Monad.Free` module. We can find out some basic information about it using PSCi, as follows:

```
> import Control.Monad.Free

> :kind Free
(Type -> Type) -> Type -> Type
```

The kind of `Free` indicates that it takes a type constructor as an argument and returns another type constructor. In fact, the `Free` monad can be used to turn any `Functor` into a `Monad`!

We begin by defining the *representation* of our monadic actions. To do this, we need to create a `Functor` with one data constructor for each monadic action we wish to support. In our case, our two monadic actions will be `elem` and `text`. We can simply modify our `Content` type as follows:

```
data ContentF a
  = TextContent String a
  | ElementContent Element a

instance Functor ContentF where
  map f (TextContent s x) = TextContent s (f x)
  map f (ElementContent e x) = ElementContent e (f x)
```

Here, the `ContentF` type constructor looks just like our old `Content` data type – however, it now takes a type argument `a`, and each data constructor has been modified to take a value of type `a` as an additional argument. The `Functor` instance simply applies the function `f` to the value of type `a` in each data constructor.

With that, we can define our new `Content` monad as a type synonym for the `Free` monad, which we construct by using our `ContentF` type constructor as the first type argument:

```
type Content = Free ContentF
```

Instead of a type synonym, we might use a `newtype` to avoid exposing the internal representation of our library to our users – by hiding the `Content` data constructor, we restrict our users to only using the monadic actions we provide.

Because `ContentF` is a `Functor`, we automatically get a `Monad` instance for `Free ContentF`.

We have to modify our `Element` data type slightly to take account of the new type argument on `Content`. We will simply require that the return type of our monadic computations be `Unit`:

```
newtype Element = Element
  { name         :: String
  , attribs      :: Array Attribute
  , content      :: Maybe (Content Unit)
  }
```

In addition, we have to modify our `elem` and `text` functions, which become our new monadic actions for the `Content` monad. To do this, we can use the `liftF` function provided by the `Control.Monad.Free` module. Here is its type:

```
liftF :: forall f a. f a -> Free f a
```

`liftF` allows us to construct an action in our free monad from a value of type `f a` for some type `a`. In our case, we can use the data constructors of our `ContentF` type constructor directly:

```
text :: String -> Content Unit
text s = liftF $ TextContent s unit

elem :: Element -> Content Unit
elem e = liftF $ ElementContent e unit
```

Some other routine modifications have to be made, but the interesting changes are in the `render` function, where we have to *interpret* our free monad.

Interpreting the Monad

The `Control.Monad.Free` module provides a number of functions for interpreting a computation in a free monad:

```
runFree
  :: forall f a
   . Functor f
  => (f (Free f a) -> Free f a)
  -> Free f a
  -> a

runFreeM
  :: forall f m a
   . (Functor f, MonadRec m)
  => (f (Free f a) -> m (Free f a))
  -> Free f a
  -> m a
```

The `runFree` function is used to compute a *pure* result. The `runFreeM` function allows us to use a monad to interpret the actions of our free monad.

Note: Technically, we are restricted to monads `m` that satisfy the stronger `MonadRec` constraint. In practice, we don't need to worry about stack overflow since `m` supports safe *monadic tail recursion*.

First, we have to choose a monad in which we can interpret our actions. We will use the `Writer String` monad to accumulate an HTML string as our result.

Our new `render` method starts by delegating to a helper function, `renderElement`, and using `execWriter` to run our computation in the `Writer` monad:

```
render :: Element -> String
render = execWriter <<< renderElement
```

`renderElement` is defined in a where block:

```
  where
    renderElement :: Element -> Writer String Unit
    renderElement (Element e) = do
```

The definition of `renderElement` is straightforward, using the `tell` action from the `Writer` monad to accumulate several small strings:

```
      tell "<"
      tell e.name
      for_ e.attribs $ \x -> do
        tell " "
        renderAttribute x
      renderContent e.content
```

Next, we define the `renderAttribute` function, which is equally simple:

```
    where
      renderAttribute :: Attribute -> Writer String Unit
      renderAttribute (Attribute x) = do
        tell x.key
        tell "=\""
        tell x.value
        tell "\""
```

The `renderContent` function is more interesting. Here, we use the `runFreeM` function to interpret the computation inside the free monad, delegating to a helper function, `renderContentItem`:

```
      renderContent :: Maybe (Content Unit) -> Writer String Unit
      renderContent Nothing = tell " />"
      renderContent (Just content) = do
        tell ">"
        runFreeM renderContentItem content
        tell "</"
        tell e.name
        tell ">"
```

The type of `renderContentItem` can be deduced from the type signature of `runFreeM`. The functor `f` is our type constructor `ContentF`, and the monad `m` is the monad in which we are interpreting the computation, namely `Writer String`. This gives the following type signature for `renderContentItem`:

```
      renderContentItem :: ContentF (Content Unit) -> Writer String (Content
Unit)
```

We can implement this function by pattern matching on the two data constructors of `ContentF`:

```
      renderContentItem (TextContent s rest) = do
        tell s
        pure rest
      renderContentItem (ElementContent e rest) = do
        renderElement e
        pure rest
```

In each case, the expression `rest` has the type `Content Unit` and represents the remainder of the interpreted computation. We can complete each case by returning the `rest` action.

That's it! We can test our new monadic API in PSCi, as follows:

```
> import Prelude
> import Data.DOM.Free
> import Effect.Console

> :paste
… log $ render $ p [] $ do
…   elem $ img [ src := "cat.jpg" ]
…   text "A cat"
… ^D

<p><img src="cat.jpg" />A cat</p>
unit
```

Exercises

1. (Medium) Add a new data constructor to the `ContentF` type to support a new action `comment`, which renders a comment in the generated HTML. Implement the new action using `liftF`. Update the interpretation `renderContentItem` to interpret your new constructor appropriately.

Extending the Language

A monad in which every action returns something of type `Unit` is not particularly interesting. In fact, aside from an arguably nicer syntax, our monad adds no extra functionality over a `Monoid`.

Let's illustrate the power of the free monad construction by extending our language with a new monadic action that returns a non-trivial result.

Suppose we want to generate HTML documents that contain hyperlinks to different sections of the document using *anchors*. We can accomplish this by generating anchor names by hand and including them at least twice in the document: once at the anchor's definition and once in each hyperlink. However, this approach has some basic issues:

- The developer might fail to generate unique anchor names.
- The developer might mistype one or more instances of the anchor name.

To protect the developer from their mistakes, we can introduce a new type that represents

anchor names and provide a monadic action for generating new unique names.

The first step is to add a new type for names:

```
newtype Name = Name String

runName :: Name -> String
runName (Name n) = n
```

Again, we define this as a newtype around `String`, but we must be careful not to export the data constructor in the module's export lists.

Next, we define an instance for the `IsValue` type class for our new type so that we can use names in attribute values:

```
instance IsValue Name where
  toValue (Name n) = n
```

We also define a new data type for hyperlinks which can appear in `a` elements, as follows:

```
data Href
  = URLHref String
  | AnchorHref Name

instance IsValue Href where
  toValue (URLHref url) = url
  toValue (AnchorHref (Name nm)) = "#" <> nm
```

With this new type, we can modify the value type of the `href` attribute, forcing our users to use our new `Href` type. We can also create a new `name` attribute, which can be used to turn an element into an anchor:

```
href :: AttributeKey Href
href = AttributeKey "href"

name :: AttributeKey Name
name = AttributeKey "name"
```

The remaining problem is that our users currently have no way to generate new names. We can provide this functionality in our `Content` monad. First, we need to add a new data constructor to our `ContentF` type constructor:

```
data ContentF a
  = TextContent String a
  | ElementContent Element a
  | NewName (Name -> a)
```

The `NewName` data constructor corresponds to an action which returns a value of type `Name`. Notice that instead of requiring a `Name` as a data constructor argument, we require the user to provide a *function* of type `Name -> a`. Remembering that the type `a` represents the *rest of the computation*, we can see that this function provides a way to continue computation after a value of type `Name` has been returned.

We also need to update the `Functor` instance for `ContentF`, taking into account the new data constructor, as follows:

```
instance Functor ContentF where
  map f (TextContent s x) = TextContent s (f x)
  map f (ElementContent e x) = ElementContent e (f x)
  map f (NewName k) = NewName (f <<< k)
```

Now we can build our new action by using the `liftF` function, as before:

```
newName :: Content Name
newName = liftF $ NewName id
```

Notice that we provide the `id` function as our continuation, meaning we return the result of type `Name` unchanged.

Finally, we need to update our interpretation function to interpret the new action. We previously used the `Writer String` monad to interpret our computations, but that monad cannot generate new names, so we must switch to something else. The `WriterT` monad transformer can be used with the `State` monad to combine the effects we need. We can define our interpretation monad as a type synonym to keep our type signatures short:

```
type Interp = WriterT String (State Int)
```

Here, the state of type `Int` will act as an incrementing counter, used to generate unique names.

Because the `Writer` and `WriterT` monads use the same type class members to abstract their actions, we do not need to change any actions – we only need to replace every reference to `Writer String` with `Interp`. However, we need to modify the handler used to run our computation. Instead of just `execWriter`, we now need to use `evalState` as well:

```
render :: Element -> String
render e = evalState (execWriterT (renderElement e)) 0
```

We also need to add a new case to `renderContentItem`, to interpret the new `NewName` data constructor:

```
renderContentItem (NewName k) = do
  n <- get
  let fresh = Name $ "name" <> show n
  put $ n + 1
  pure (k fresh)
```

Here, we are given a continuation `k` of type `Name -> Content a`, and we need to construct an interpretation of type `Content a`. Our interpretation is simple: we use `get` to read the state, use that state to generate a unique name, then use `put` to increment the state. Finally, we pass our new name to the continuation to complete the computation.

With that, we can try out our new functionality in PSCi, by generating a unique name inside the `Content` monad and using it as both the name of an element and the target of a hyperlink:

```
> import Prelude
> import Data.DOM.Name
> import Effect.Console

> :paste
… render $ p [ ] $ do
…   top <- newName
…   elem $ a [ name := top ] $
…     text "Top"
…   elem $ a [ href := AnchorHref top ] $
…     text "Back to top"
… ^D

<p><a name="name0">Top</a><a href="#name0">Back to top</a></p>
unit
```

You can verify that multiple calls to `newName` do, in fact, result in unique names.

Exercises

1. (Medium) We can simplify the API further by hiding the `Element` type from its users. Make these changes in the following steps:

    - Combine functions like `p` and `img` (with return type `Element`) with the `elem` action to create new actions with return type `Content Unit`.
    - Change the `render` function to accept an argument of type `Content Unit` instead of `Element`.

2. (Medium) Hide the implementation of the `Content` monad using a `newtype` instead of a type synonym. You should not export the data constructor for your `newtype`.

3. (Difficult) Modify the `ContentF` type to support a new action

   ```
   isMobile :: Content Boolean
   ```

   which returns a boolean value indicating whether or not the document is being rendered for display on a mobile device.

   Hint: use the `ask` action and the `ReaderT` monad transformer to interpret this action. Alternatively, you might prefer to use the `RWS` monad.

Conclusion

In this chapter, we developed a domain-specific language for creating HTML documents by incrementally improving a naive implementation using some standard techniques:

- We used *smart constructors* to hide the details of our data representation, only permitting the user to create documents that were *correct-by-construction*.
- We used a *user-defined infix binary operator* to improve the syntax of the language.
- We used *phantom types* to encode additional information in the types of our data, preventing the user from providing attribute values of the wrong type.
- We used the *free monad* to turn our array representation of a collection of content into a monadic representation supporting do notation. We then extended this representation to support a new monadic action and interpreted the monadic computations using standard monad transformers.

These techniques all leverage PureScript's module and type systems, either to prevent the user from making mistakes or to improve the syntax of the domain-specific language.

Implementing domain-specific languages in functional programming languages is an area of active research. Still, hopefully, this provides a useful introduction to some simple techniques and illustrates the power of working in a language with expressive types.