# PureScript by Example

Functional Programming for the Web

Phil Freeman

# PureScript by Example

## Functional Programming for the Web

Phil Freeman

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

This version was published on 2017-09-24

# Tweet This Book!

Please help Phil Freeman by spreading the word about this book on Twitter!

The suggested hashtag for this book is #purescript.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#purescript

# Contents

# 1. Introduction

## 1.1 Functional JavaScript

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

- Libraries such as UnderscoreJS[1] 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.

```
require('fs').readFile(sourceFile, function (error, data) {
  if (!error) {
    require('fs').writeFile(destFile, data, function (error) {
      if (!error) {
        console.log("File copied");
      }
    });
  }
});
```

- Libraries such as React[2] and virtual-dom[3] model views as pure functions of application state.

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

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

[1] http://underscorejs.org

[2] http://facebook.github.io/react/

[3] https://github.com/Matt-Esch/virtual-dom

## 1.2 Types and Type Inference

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

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

```
iAmANumber =
  let square x = x * x
  in square 42.0
```

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

```
iterate f 0 x = x
iterate f n x = iterate f (n - 1) (f x)
```

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

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

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

## 1.3 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.

## 1.4 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 `0.11.*` of the PureScript compiler.

## 1.5 About You

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

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

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

## 1.6 How to Read This Book

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

Each chapter will focus on a single practical example, providing the motivation for any new ideas introduced. Code for each chapter are available from the book's GitHub repository[4]. 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 Control.Monad.Eff.Console (log)

main = log "Hello, World!"
```

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

```
$ pulp 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.

[4] https://github.com/paf31/purescript-book

## 1.7 Getting Help

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

- The PureScript IRC channel is a great place to chat about issues you may be having. Point your IRC client at irc.freenode.net, and connect to the #purescript channel.
- The PureScript website[5] contains links to several learning resources, including code samples, videos and other resources for beginners.
- The PureScript documentation repository[6] collects articles and examples on a wide variety of topics, written by PureScript developers and users.
- Try PureScript![7] is a website which allows users to compile PureScript code in the web browser, and contains several simple examples of code.
- Pursuit[8] is a searchable database of PureScript types and functions.

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

## 1.8 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[9], and can be reached on Twitter[10].

## 1.9 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.

[5] http://purescript.org
[6] https://github.com/purescript/documentation
[7] http://try.purescript.org
[8] http://pursuit.purescript.org
[9] http://blog.functorial.com
[10] http://twitter.com/paf31

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[11].

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

[11] https://creativecommons.org/licenses/by/4.0/

# 2. Getting Started

## 2.1 Chapter Goals

In this chapter, the goal will be to set up a working PureScript development environment, and to write our first PureScript program.

Our first project will be a very simple PureScript library, which will provide a single function which can compute the length of the diagonal in a right-angled triangle.

## 2.2 Introduction

Here are the tools we will be using to set up our PureScript development environment:

- `purs`[1] - The PureScript compiler itself.
- `npm`[2] - The Node Package Manager, which will allow us to install the rest of our development tools.
- Pulp[3] - A command-line tool which automates many of the tasks associated with managing PureScript projects.

The rest of the chapter will guide you through installing and configuring these tools.

## 2.3 Installing PureScript

The recommended approach to installing the PureScript compiler is to download a binary release for your platform from the PureScript website[4].

You should verify that the PureScript compiler executables are available on your path. Try running the PureScript compiler on the command line to verify this:

```
$ purs
```

Other options for installing the PureScript compiler include:

- Via NPM: `npm install -g purescript`.
- Building the compiler from source. Instructions can be found on the PureScript website.

[1] http://purescript.org

[2] http://npmjs.org

[3] https://github.com/bodil/pulp

[4] http://purescript.org

## 2.4 Installing Tools

If you do not have a working installation of NodeJS[5], you should install it. This should also install the `npm` package manager on your system. Make sure you have `npm` installed and available on your path.

You will also need to install the Pulp command line tool, and the Bower package manager using `npm`, as follows:

```
$ npm install -g pulp bower
```

This will place the `pulp` and `bower` command line tools on your path. At this point, you will have all the tools needed to create your first PureScript project.

## 2.5 Hello, PureScript!

Let's start out simple. We'll use Pulp to compile and run a simple Hello World! program.

Begin by creating a project in an empty directory, using the `pulp init` command:

```
$ mkdir my-project
$ cd my-project
$ pulp init

* Generating project skeleton in ~/my-project

$ ls

bower.json      src             test
```

Pulp has created two directories, `src` and `test`, and a `bower.json` configuration file for us. The `src` directory will contain our source files, and the `test` directory will contain our tests. We will use the `test` directory later in the book.

Modify the `src/Main.purs` file to contain the following content:

```
module Main where

import Control.Monad.Eff.Console

main = log "Hello, World!"
```

This small sample illustrates a few key ideas:

[5] http://nodejs.org/

- Every file begins with a module header. A module name consists of one or more capitalized words separated by dots. In this case, only a single word is used, but `My.First.Module` would be an equally valid module name.
- Modules are imported using their full names, including dots to separate the parts of the module name. Here, we import the `Control.Monad.Eff.Console` module, which provides the `log` function.
- The `main` program is defined as a function application. In PureScript, function application is indicated with whitespace separating the function name from its arguments.

Let's build and run this code using the following command:

```
$ pulp run

* Building project in ~/my-project
* Build successful.
Hello, World!
```

Congratulations! You just compiled and executed your first PureScript program.

## 2.6 Compiling for the Browser

Pulp can be used to turn our PureScript code into Javascript suitable for use in the web browser, by using the `pulp browserify` command:

```
$ pulp browserify

* Browserifying project in ~/my-project
* Building project in ~/my-project
* Build successful.
* Browserifying...
```

Following this, you should see a large amount of Javascript code printed to the console. This is the output of the Browserify[6] tool, applied to a standard PureScript library called the *Prelude*, as well as the code in the `src` directory. This Javascript code can be saved to a file, and included in a HTML document. If you try this, you should see the words "Hello, World!" printed to your browser's console.

## 2.7 Removing Unused Code

Pulp provides an alternative command, `pulp build`, which can be used with the `-O` option to apply *dead code elimination*, which removes unnecessary Javascript from the output. The result is much smaller:

[6] http://browserify.org/

```
$ pulp build -O --to output.js

* Building project in ~/my-project
* Build successful.
* Bundling Javascript...
* Bundled.
```

Again, the generated code can be used in a HTML document. If you open `output.js`, you should see a few compiled modules which look like this:

```
(function(exports) {
  "use strict";

  var Control_Monad_Eff_Console = PS["Control.Monad.Eff.Console"];

  var main = Control_Monad_Eff_Console.log("Hello, World!");
  exports["main"] = main;
})(PS["Main"] = PS["Main"] || {});
```

This illustrates a few points about the way the PureScript compiler generates Javascript code:

- Every module gets turned into an object, created by a wrapper function, which contains the module's exported members.
- PureScript tries to preserve the names of variables wherever possible
- Function applications in PureScript get turned into function applications in JavaScript.
- The main method is run after all modules have been defined, and is generated as a simple method call with no arguments.
- PureScript code does not rely on any runtime libraries. All of the code that is generated by the compiler originated in a PureScript module somewhere which your code depended on.

These points are important, since they mean that PureScript generates simple, understandable code. In fact, the code generation process in general is quite a shallow transformation. It takes relatively little understanding of the language to predict what JavaScript code will be generated for a particular input.

## 2.8 Compiling CommonJS Modules

Pulp can also be used to generate CommonJS modules from PureScript code. This can be useful when using NodeJS, or just when developing a larger project which uses CommonJS modules to break code into smaller components.

To build CommonJS modules, use the `pulp build` command (without the `-O` option):

```
$ pulp build

* Building project in ~/my-project
* Build successful.
```

The generated modules will be placed in the `output` directory by default. Each PureScript module will be compiled to its own CommonJS module, in its own subdirectory.

## 2.9 Tracking Dependencies with Bower

To write the `diagonal` function (the goal of this chapter), we will need to be able to compute square roots. The `purescript-math` package contains type definitions for functions defined on the JavaScript `Math` object, so let's install it:

```
$ bower install purescript-math --save
```

The `--save` option causes the dependency to be added to the `bower.json` configuration file.

The `purescript-math` library sources should now be available in the `bower_components` subdirectory, and will be included when you compile your project.

## 2.10 Computing Diagonals

Let's write the `diagonal` function, which will be an example of using a function from an external library.

First, import the `Math` module by adding the following line at the top of the `src/Main.purs` file:

```
import Math (sqrt)
```

It's also necessary to import the `Prelude` module, which defines very basic operations such as numeric addition and multiplication:

```
import Prelude
```

Now define the `diagonal` function as follows:

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

Note that there is no need to define a type for our function. The compiler is able to infer that `diagonal` is a function which takes two numbers and returns a number. In general, however, it is a good practice to provide type annotations as a form of documentation.

Let's also modify the `main` function to use the new `diagonal` function:

```
main = logShow (diagonal 3.0 4.0)
```

Now compile and run the project again, using `pulp run`:

```
$ pulp run

* Building project in ~/my-project
* Build successful.
5.0
```

## 2.11 Testing Code Using the Interactive Mode

The PureScript compiler also ships with an interactive REPL called PSCi. This can be very useful for testing your code, and experimenting with new ideas. Let's use PSCi to test the `diagonal` function.

Pulp can load source modules into PSCi automatically, via the `pulp repl` command:

```
$ pulp repl
>
```

You can type `:?` to see a list of commands:

```
> :?
The following commands are available:

    :?                        Show this help menu
    :quit                     Quit PSCi
    :reset                    Reset
    :browse      <module>     Browse <module>
    :type        <expr>       Show the type of <expr>
    :kind        <type>       Show the kind of <type>
    :show        import       Show imported modules
    :show        loaded       Show loaded modules
    :paste       paste        Enter multiple lines, terminated by ^D
```

By pressing the Tab key, you should be able to see a list of all functions available in your own code, as well as any Bower dependencies and the Prelude modules.

Start by importing the `Prelude` module:

```
> import Prelude
```

Try evaluating a few expressions now:

```
> 1 + 2
3

> "Hello, " <> "World!"
"Hello, World!"
```

Let’s try out our new `diagonal` function in PSCi:

```
> import Main
> diagonal 5.0 12.0

13.0
```

You can also use PSCi to define functions:

```
> double x = x * 2

> double 10
20
```

Don’t worry if the syntax of these examples is unclear right now - it will make more sense as you read through the book.

Finally, you can check the type of an expression by using the `:type` command:

```
> :type true
Boolean

> :type [1, 2, 3]
Array Int
```

Try out the interactive mode now. If you get stuck at any point, simply use the Reset command `:reset` to unload any modules which may be compiled in memory.

## Exercises

1. (Easy) Use the `pi` constant, which is defined in the `Math` module, to write a function `circleArea` which computes the area of a circle with a given radius. Test your function using PSCi (*Hint*: don’t forget to import `pi` by modifying the `import Math` statement).
2. (Medium) Use `bower install` to install the `purescript-globals` package as a dependency. Test out its functions in PSCi (*Hint*: you can use the `:browse` command in PSCi to browse the contents of a module).

## 2.12 Conclusion

In this chapter, we set up a simple PureScript project using the Pulp tool.

We also wrote our first PureScript function, and a JavaScript program which could be compiled and executed either in the browser or in NodeJS.

We will use this development setup in the following chapters to compile, debug and test our code, so you should make sure that you are comfortable with the tools and techniques involved.

# 3. Functions and Records

## 3.1 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.

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

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

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

```
$ cd chapter3
$ bower update
$ pulp build
```

## 3.3 Simple Types

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

```
$ pulp repl

> :type 1.0
Number

> :type "test"
String

> :type true
Boolean
```

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

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

```
> :type 1
Int
```

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

```
> :type 'a'
Char
```

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

```
> :type [1, 2, 3]
Array Int

> :type [true, false]
Array Boolean

> :type [1, false]
Could not match type Int with Boolean.
```

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

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

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

> :type author
{ name :: String
, interests :: Array String
}
```

This type indicates that the specified object has two *fields*, a `name` field which has type `String`, and an `interests` field, which has type `Array String`, i.e. an array of `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. The PureScript standard libraries provide plenty of examples of functions, and we will see more in this chapter:

```
> import Prelude
> :type flip
forall a b c. (a -> b -> c) -> b -> a -> c

> :type const
forall a b. a -> b -> a
```

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

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

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

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

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

```
> add 10 20
30
```

## 3.4 Quantified Types

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

```
> :type flip
forall a b c. (a -> b -> c) -> b -> a -> c
```

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

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

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

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

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

"10Ten"
```

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

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

Could not match type Int with type String
```

## 3.5 Notes On Indentation

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

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

Therefore, the following is valid PureScript code:

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

But this is not valid code:

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

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

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

```
> :paste
… x = 1
… y = 2
… ^D
```

but this is not:

```
> :paste
… x = 1
…  y = 2
… ^D
```

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

```
example x y z = foo + bar
  where
    foo = x * y
    bar = y * z
```

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

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

## 3.6 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 type `String`, and the `address` field will have 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` is not the same as `Array Entry`, which represents an *array* of entries.

## 3.7 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.

## 3.8 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, which 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.

## 3.9 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:

```
$ pulp build
```

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

```
$ pulp repl

> import Data.AddressBook
```

We can create an entry by using a record literal, which looks just like an anonymous object in JavaScript. Bind it to a name with a `let` expression:

```
> 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"
```

## 3.10 Creating Address Books

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

```
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, and an AddressBook as a second argument, and returns a new AddressBook.

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

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

```
$ pulp repl

> import Data.List
> :type Cons

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

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

```
Entry -> List Entry -> List Entry
```

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

```
Entry -> AddressBook -> AddressBook
```

In our case, we already have the appropriate inputs: an `Entry`, and a `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.

## 3.11 Curried Functions

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

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

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

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

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

```
> :type insertEntry entry

AddressBook -> AddressBook
```

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

```
> :type (insertEntry entry) emptyBook
AddressBook
```

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

```
> :type insertEntry example emptyBook
AddressBook
```

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

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 another function.

Now consider the definition of `insertEntry`:

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

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

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

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

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

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

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

## 3.12 Querying the Address Book

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

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

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

```
$ pulp repl

> import Data.List
> :type filter

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

> :type head

forall a. List a -> Maybe a
```

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

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

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

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

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

head :: AddressBook -> Maybe Entry
```

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

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

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

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

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

And here is the definition of `findEntry`:

```
findEntry firstName lastName book = head $ filter filterEntry book
  where
    filterEntry :: Entry -> Boolean
    filterEntry entry = entry.firstName == firstName && entry.lastName == lastName
```

Let's go over this code step by step.

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

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

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

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

## 3.13 Infix Function Application

In the code for `findEntry` above, we used a different form of function application: the `head` function was applied to the expression `filter filterEntry book` by using the infix `$` symbol.

This is equivalent to the usual application `head (filter filterEntry book)`

`($)` is just an alias for a regular function called `apply`, which is defined in the Prelude. It is defined as follows:

```
apply :: forall a b. (a -> b) -> a -> b
apply f x = f x

infixr 0 apply as $
```

So `apply` takes a function and a value and applies the function to the value. The `infixr` keyword is used to define `($)` as an alias for `apply`.

But why would we want to use `$` instead of regular function application? The reason is that `$` is a right-associative, low precedence operator. This means that `$` allows us to remove sets of parentheses for deeply-nested applications.

For example, the following nested function application, which finds the street in the address of an employee's boss:

```
street (address (boss employee))
```

becomes (arguably) easier to read when expressed using `$`:

```
street $ address $ boss employee
```

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

## 3.15 Tests, Tests, Tests …

Now that we have the core of a working application, let's try it out using PSCi.

```
$ pulp repl
```

```
> import Data.AddressBook
```

Let's first try looking up an entry in the empty address book (we obviously expect this to return an empty result):

```
> findEntry "John" "Smith" emptyBook

No type class instance was found for

    Data.Show.Show { firstName :: String
                   , lastName :: String
                   , address :: { street :: String
                                , city :: String
                                , state :: String
                                }
                   }
```

An error! Not to worry, this error simply means that PSCi doesn't know how to print a value of type `Entry` as a String.

The return type of `findEntry` is `Maybe Entry`, which we can convert to a `String` by hand.

Our `showEntry` function expects an argument of type `Entry`, but we have a value of type `Maybe Entry`. Remember that this means that the function returns an optional value of type `Entry`. What we need to do is apply the `showEntry` function if the optional value is present, and propagate the missing value if not.

Fortunately, the Prelude module provides a way to do this. The `map` operator can be used to lift a function over an appropriate type constructor like `Maybe` (we'll see more on this function, and others like it, later in the book, when we talk about functors):

```
> import Prelude
> map showEntry (findEntry "John" "Smith" emptyBook)

Nothing
```

That's better - the return value `Nothing` indicates that the optional return value does not contain a value - just as we expected.

For ease of use, we can create a function which prints an `Entry` as a String, so that we don't have to use `showEntry` every time:

```
> printEntry firstName lastName book = map showEntry (findEntry firstName lastName book)
```

Now let's create a non-empty address book, and try again. We'll reuse our example entry from earlier:

```
> book1 = insertEntry entry emptyBook

> printEntry "John" "Smith" book1

Just ("Smith, John: 123 Fake St., Faketown, CA")
```

This time, the result contained the correct value. Try defining an address book `book2` with two names by inserting another name into `book1`, and look up each entry by name.

## 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`.
2. (Medium) Write a function which looks up an `Entry` given a street address, by reusing the existing code in `findEntry`. Test your function in PSCi.
3. (Medium) Write a function which tests whether a name appears in a `AddressBook`, returning a Boolean value. *Hint*: Use PSCi to find the type of the `Data.List.null` function, which test whether a list is empty or not.
4. (Difficult) Write a function `removeDuplicates` which removes duplicate address book entries with the same first and last names. *Hint*: Use PSCi to find the type of the `Data.List.nubBy` function, which removes duplicate elements from a list based on an equality predicate.

## 3.16 Conclusion

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

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

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

# 4. Recursion, Maps And Folds

## 4.1 Chapter Goals

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

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

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

## 4.2 Project Setup

The source code for this chapter is contained in the two files `src/Data/Path.purs` and `src/FileOperations.purs`.

The `Data.Path` module contains a model of a virtual filesystem. You do not need to modify the contents of this module.

The `FileOperations` module contains functions which use the `Data.Path` API. Solutions to the exercises can be completed in this file.

The project has the following Bower dependencies:

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

## 4.3 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:

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

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

Here is another common example, which computes the *Fibonnacci function*:

```
fib :: Int -> Int
fib 0 = 1
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.

## 4.4 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)
import Data.Array.Partial (tail)
import Partial.Unsafe (unsafePartial)

length :: forall a. Array a -> Int
length arr =
  if null arr
    then 0
    else 1 + length (unsafePartial tail arr)
```

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

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

### Exercises

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

## 4.5 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:

```
$ pulp repl

> import Prelude
> map (\n -> n + 1) [1, 2, 3, 4, 5]
[2, 3, 4, 5, 6]
```

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

## 4.6 Infix Operators

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

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

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

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

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

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

```
> :type map
forall a b f. Functor f => (a -> b) -> f a -> f b
```

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

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

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

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

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

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

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

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

```
infix 8 range as ..
```

We can use this operator as follows:

```
> import Data.Array

> 1 .. 5
[1, 2, 3, 4, 5]

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

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

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

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

If we wanted to assign an *associativity* (left or right) to an infix operator, we could do so with the `infixl` and `infixr` keywords instead.

## 4.7 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) Use the `map` or `<$>` function to write a function which calculates the squares of an array of numbers.
2. (Easy) Use the `filter` function to write a function which removes the negative numbers from an array of numbers.
3. (Medium) Define an infix synonym `<$?>` for `filter`. Rewrite your answer to the previous question to use your new operator. Experiment with the precedence level and associativity of your operator in PSCi.

## 4.8 Flattening Arrays

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

```
> import Data.Array

> :type concat
forall a. Array (Array a) -> Array a

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

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

Let's see it in action:

```
> import Data.Array

> :type concatMap
forall a b. (a -> Array b) -> Array a -> Array b

> concatMap (\n -> [n, n * n]) (1 .. 5)
[1,1,2,4,3,9,4,16,5,25]
```

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

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

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

## 4.9 Array Comprehensions

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

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

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

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

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

We can test our function

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

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

```
> :paste
… pairs' n =
…   concatMap (\i ->
…     map (\j -> [i, j]) (1 .. n)
…   ) (1 .. n)
… ^D

> pairs' 3
[[1,1],[1,2],[1,3],[2,1],[2,2],[2,3],[3,1],[3,2],[3,3]]
```

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

```
> :paste
… pairs'' n =
…   concatMap (\i ->
…     map (\j -> [i, j]) (i .. n)
…   ) (1 .. n)
… ^D
> pairs'' 3
[[1,1],[1,2],[1,3],[2,2],[2,3],[3,3]]
```

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

```
> import Data.Foldable

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

> factors 10
[[1,10],[2,5]]
```

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

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

## 4.10 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 which uses do notation. The block consists of expressions of a few types:

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

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

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

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

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

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

and the result would be the same.

## 4.11 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.MonadZero module (from the purescript-control package):

```
import Control.MonadZero (guard)

factors :: Int -> Array (Array Int)
factors 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.MonadZero

> :type guard
forall m. MonadZero m => Boolean -> m Unit
```

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

```
Boolean -> Array Unit
```

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

```
> import Data.Array

> length $ guard true
1

> length $ guard false
0
```

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

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

## Exercises

1. (Easy) Use the `factors` function to define a function `isPrime` which tests if its integer argument is prime or not.
2. (Medium) Write a function 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) A *Pythagorean triple* is an array of numbers `[a, b, c]` such that $a^2 + b^2 = c^2$. Use the `guard` function in an array comprehension to write a function `triples` which takes a number `n` and calculates all Pythagorean triples whose components are less than `n`. Your function should have type `Int -> Array (Array Int)`.
4. (Difficult) Write a function `factorizations` which produces all *factorizations* of an integer `n`, i.e. arrays of integers whose product is `n`. *Hint*: for an integer greater than `1`, break the problem down into two subproblems: finding the first factor, and finding the remaining factors.

## 4.12 Folds

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

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

```
> import Data.Foldable

> :type foldl
forall a b f. Foldable f => (b -> a -> b) -> b -> f a -> b

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

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

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

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

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

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

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

```
> foldl (+) 0 (1 .. 5)
15
```

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

```
> foldr (+) 0 (1 .. 5)
15
```

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

```
> foldl (\acc n -> acc <> show n) "" [1,2,3,4,5]
"12345"

> foldr (\n acc -> acc <> show n) "" [1,2,3,4,5]
"54321"
```

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

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

whereas the right fold is equivalent to this:

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

## 4.13 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:

```
> f 0 = 0
> f n = 1 + f (n - 1)

> f 10
10

> f 100000
RangeError: Maximum call stack size exceeded
```

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

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

*Note*: more complete solutions to the problem can be implemented in libraries using so-called *trampolining*, but that is beyond the scope of this chapter. The interested reader can consult the documentation for the `purescript-free` and `purescript-tailrec` packages.

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

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

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

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

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

## 4.14 Accumulators

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

For example, consider this array recursion which reverses the input array by appending elements at the head of the input array to the end of the result.

```
reverse :: forall a. Array a -> Array a
reverse [] = []
reverse xs = snoc (reverse (unsafePartial tail xs))
                  (unsafePartial head xs)
```

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:

```
reverse :: forall a. Array a -> Array a
reverse = reverse' []
  where
    reverse' acc [] = acc
    reverse' acc xs = reverse' (unsafePartial head xs : acc)
                               (unsafePartial tail xs)
```

In this case, we delegate to the helper function `reverse'`, which performs the heavy lifting of reversing the array. Notice though that the function `reverse'` is tail recursive - its only recursive call is in the last case, and is in tail position. This means that the generated code will be a *while loop*, and will not blow the stack for large inputs.

To understand the second implementation of `reverse`, note that the helper function `reverse'` essentially uses the accumulator parameter to maintain an additional piece of state - the partially constructed result. The result starts out empty, and grows by one element for every element in the input array. However, because later elements are added at the front of the array, the result is the original array in reverse!

Note also that while we might think of the accumulator as “state”, there is no direct mutation going on. The accumulator is an immutable array, and we simply use function arguments to thread the state through the computation.

## 4.15 Prefer Folds to Explicit Recursion

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

For example, the `reverse` example can be written as a fold in at least two ways. Here is a version which uses `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) Use `foldl` to test whether an array of boolean values are all true.
2. (Medium) Characterize those arrays `xs` for which the function `foldl (==) false xs` returns true.
3. (Medium) Rewrite the following function in tail recursive form using an accumulator parameter:

```
import Prelude
import Data.Array.Partial (head, tail)

count :: forall a. (a -> Boolean) -> Array a -> Int
count _ [] = 0
count p xs = if p (unsafePartial head xs)
               then count p (unsafePartial tail xs) + 1
               else count p (unsafePartial tail xs)
```

4. (Medium) Write `reverse` in terms of `foldl`.

## 4.16 A Virtual Filesystem

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

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

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

isDirectory :: Path -> Boolean
```

We can try out the API in PSCi:

```
$ pulp repl

> import Data.Path

> root
/

> isDirectory root
true

> ls root
[/bin/,/etc/,/home/]
```

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

## 4.17 Listing All Files

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

```
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 at the same time.

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

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

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

Let's try this function in PSCi:

```
> import FileOperations
> import Data.Path

> allFiles root

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

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

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

Here is the new version:

```
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 fold to determine the largest and smallest files in the filesystem.
3. (Difficult) 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 "/bin/ls"
   Just (/bin/)

   > whereIs "/bin/cat"
   Nothing
   ```

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

## 4.18 Conclusion

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

# 5. Pattern Matching

## 5.1 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 enable a similar level of expressiveness in the language of types - they are closely related to pattern matching.

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

## 5.2 Project Setup

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

The project uses some Bower packages which we have already seen, and adds the following new dependencies:

- `purescript-globals`, which provides access to some common JavaScript values and functions.
- `purescript-math`, which provides access to the JavaScript `Math` module.

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)
```

The `Data.Picture` module also imports the `Global` and `Math` modules, but this time using the `as` keyword:

```
import Global as Global
import Math as Math
```

This makes the types and functions in those modules available for use, but only by using *qualified names*, like `Global.infinity` and `Math.max`. This can be useful to avoid overlapping imports, or just to make it clearer which modules certain things are imported from.

*Note*: it is not necessary to use the same module name as the original module for a qualified import. Shorter qualified names like `import Math as M` are possible, and quite common.

## 5.3 Simple Pattern Matching

Let's begin by looking at an example. Here is a function which 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 which look a lot like the code above. This is one benefit of pattern matching: it allows you to define code by cases, writing simple, declarative code which looks like a specification of a mathematical function.

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 kinds of patterns, we will see that different types of patterns correspond to different ways to choose names from the input arguments.

## 5.4 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 (_), which match any argument, and which do not bind any names.

Here are two more examples which 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.

## 5.5 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 which must be satisfied in addition to the constraints imposed by the patterns. Here is the Euclidean algorithm rewritten to use a guard:

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

As this example demonstrates, 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 cases zero and non-zero inputs.
2. (Medium) Look up *Pascal's Rule* for computing binomial coefficients. Use it to write a function which computes binomial coefficients using pattern matching.

## 5.6 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 which matches arrays of length five, binding each of its five elements in a different way:

```
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, but 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 which 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.

## 5.7 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. { 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 are able to 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 we could have also written

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

and PSCi would have inferred the same type.

We will see row polymorphism again later, when we discuss *extensible effects*.

## 5.8 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, but 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
```

## 5.9 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.

### 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, taking into account row polymorphism? What about the `livesInLA` function defined above?
3. (Medium) Write a function `fromSingleton` which 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`

## 5.10 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 "longest zero suffix" of an array (the longest suffix which sums to zero):

```
import Data.Array.Partial (tail)
import Partial.Unsafe (unsafePartial)

lzs :: Array Int -> Array Int
lzs [] = []
lzs xs = case sum xs of
           0 -> xs
           _ -> lzs (unsafePartial 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 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.

## 5.11 Pattern Match Failures and Partial Functions

If patterns in a case expression are tried in order, then what happens in the case 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 which return a value for any combination of inputs are called *total* functions, and functions which 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 with type `Maybe a` for some `a`, using `Nothing` to indicate failure. 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 that 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 which 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 would have been matched by a prior case):

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

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

```
Redundant cases have been detected.
The definition has the following redundant 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 `pulp build`.

## 5.12 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 shape types: 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 sort of 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
```

The `Point` type might also be defined as an algebraic data type, as follows:

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

The `Point` data type illustrates some interesting points:

- 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.
- Even though ADTs are useful for describing data with multiple constructors, they can also be useful when there is only a single constructor.
- The constructors of an algebraic data type might have the same name as the ADT itself. This is quite common, and it is important not to confuse the `Point` *type constructor* with the `Point` *data constructor* - they live in different namespaces.

This declaration defines `Shape` as a sum of different constructors, and for each constructor identifies the data that is included. A `Shape` is either a `Circle` which 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 are defined after the equals symbol, and are separated by pipe characters (`|`).

Let's see another example from PureScript's standard libraries. We saw the `Maybe` type, which is used to to define optional values, earlier in the book. Here is it's definition from the `purescript-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`".

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 `purescript-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.

## 5.13 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 = Point { x: 0.0, y: 0.0 }

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

To construct the points `p1` and `p2`, we apply the `Point` constructor to its single argument, which is a record.

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:

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

showShape :: Shape -> String
showShape (Circle c r)      = ...
showShape (Rectangle c w h) = ...
showShape (Line start end)  = ...
showShape (Text p text) = ...
```

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.

`showPoint` is another example of pattern matching. In this case, there is only a single case, but we use a nested pattern to match the fields of the record contained inside the `Point` constructor.

## 5.14 Record Puns

The `showPoint` function matches a record inside its argument, binding the `x` and `y` properties to values with the same names. In PureScript, we can simplify this sort of pattern match as follows:

```
showPoint :: Point -> String
showPoint (Point { x, y }) = ...
```

Here, we only specify the names of the properties, 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 `x` and `y` in scope, we can construct a `Point` using `Point { x, y }`:

```
origin :: Point
origin = Point { x, y }
  where
    x = 0.0
    y = 0.0
```

This can be useful for improving readability of code in some circumstances.

## Exercises

1. (Easy) Construct a value of type `Shape` which represents a circle centered at the origin with radius `10.0`.
2. (Medium) Write a function from `Shape`s to `Shape`s, which scales its argument by a factor of `2.0`, center the origin.
3. (Medium) Write a function 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`.

## 5.15 Newtypes

There is an important 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. 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 pixels and inches:

```
newtype Pixels = Pixels Number
newtype Inches = Inches Number
```

This way, it is impossible to pass a value of type `Pixels` to a function which expects `Inches`, but there is no runtime performance overhead.

Newtypes will become important when we cover *type classes* in the next chapter, since they allow us to attach different behavior to a type without changing its representation at runtime.

## 5.16 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 `pulp build` and open PSCi with `pulp repl`:

```
$ pulp build
$ pulp repl

> import Data.Picture

> :paste
… showPicture
…   [ Line (Point { x: 0.0, y: 0.0 })
…          (Point { x: 1.0, y: 1.0 })
…   ]
… ^D

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

## 5.17 Computing Bounding Rectangles

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

The `Bounds` data type defines a bounding rectangle. It is also defined as an algebraic data type with a single constructor:

```
data Bounds = 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` which computes the area of a `Shape`. For the purposes of this exercise, the area of 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.

## 5.18 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 which allows many idiomatic JavaScript functions to be given a type. We will see this idea again later in the book.

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.

# 6. Type Classes

## 6.1 Chapter Goals

This chapter will introduce a powerful form of abstraction which is 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 allow 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.

## 6.2 Project Setup

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

The project has the following Bower dependencies:

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

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

## 6.3 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 showBoolean :: Show Boolean where
  show true = "true"
  show false = "false"
```

This code declares a type class instance called `showBoolean` - in PureScript, type class instances are named to aid the readability of the generated JavaScript. We say that the `Boolean` type *belongs to the `Show` type class*.

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\")"
```

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 was unable to 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)
```

### Exercises

1. (Easy) Use the showShape function from the previous chapter to define a Show instance for the Shape type.

## 6.4 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 just an alias for eq.

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

Note that 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 which support ordering. The comparison operators `<` and `>` along with their non-strict companions `<=` and `>=`, can be defined in terms of `compare`.

```
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 which 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 is beyond the scope of this chapter. The interested reader is encouraged to read the documentation for the superclasses of `Field` in `purescript-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. Several other standard instances are provided by the `purescript-monoid` package.

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

The `Monoid` type class (provided by the `purescript-monoid` 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 which concatenates an array of values in some monoid by using a fold. In PSCi:

```
> 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 `purescript-monoid` 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 `purescript-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 that 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 which 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 which are foldable. `purescript-foldable-traversable` also defines `Foldable` instances for types like `Maybe` and `Tuple`, and other libraries like `purescript-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 which 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 id 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 thing 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.

## Exercises

1. (Easy) The following newtype represents a complex number:

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

   Define `Show` and `Eq` instances for `Complex`.

## 6.5 Type Annotations

Types of functions can be constrained by using type classes. Here is an example: suppose we want to write a function which 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. 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`s.

## 6.6 Overlapping Instances

PureScript has another rule regarding type class instances, called the *overlapping instances rule*. Whenever a type class instance is required at a function call site, PureScript will use the information inferred by the type checker to choose the correct instance. At that time, there should be exactly one appropriate instance for that type. If there are multiple valid instances, the compiler will issue a warning.

To demonstrate this, we can try creating two conflicting type class instances for an example type. In the following code, we create two overlapping `Show` instances for the type `T`:

```
module Overlapped where

import Prelude

data T = T

instance showT1 :: Show T where
  show _ = "Instance 1"

instance showT2 :: Show T where
  show _ = "Instance 2"
```

This module will compile with no warnings. However, if we *use* `show` at type `T` (requiring the compiler to to find a `Show` instance), the overlapping instances rule will be enforced, resulting in a warning:

```
Overlapping instances found for Prelude.Show T
```

The overlapping instances rule is enforced so that automatic selection of type class instances is a predictable process. If we allowed two type class instances for a type to exist, then either could be chosen depending on the order of module imports, and that could lead to unpredictable behavior of the program at runtime, which is undesirable.

If it is truly the case that there are two valid type class instances for a type, satisfying the appropriate laws, then a common approach is to define newtypes which wrap the existing type. Since different newtypes are allowed to have different type class instances under the overlapping instances rule, there is no longer an issue. This approach is taken in PureScript's standard libraries, for example in `purescript-maybe`, where the `Maybe a` type has multiple valid instances for the `Monoid` type class.

## 6.7 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 showArray :: 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 showEither :: (Show a, Show b) => Show (Either a b) where
  ...
```

These two type class instances are provided in the `purescript-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` which reuses the instances for `Eq a` and `Eq (Array a)`.
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 which is greater than any other value:

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

   Write an `Ord` instance for `Extended a` which reuses the `Ord` instance for `a`.
5. (Difficult) Write a `Foldable` instance for `NonEmpty`. *Hint*: reuse the `Foldable` instance for arrays.
6. (Difficult) Given an type constructor `f` which defines an ordered container (and so has a `Foldable` instance), we can create a new container type which includes an extra element at the front:

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

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

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

## 6.8 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 in fact, 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 as String

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

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

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

The `Stream` module defines a class `Stream` which identifies types which 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 which work over arbitrary streams. For example, here is a function which 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`.

## 6.9 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 is unable to make that deduction automatically, and cannot 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 element. Stream stream element => stream -> Maybe stream

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

Functional dependencies can be quite useful when using multi-parameter type classes to design certain APIs.

## 6.10 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 global properties of our code in the type system.

An important example is the `Partial` class which we saw earlier when discussing partial functions. We've seen the partial functions `head` and `tail`, defined in `Data.Array.Partial` already:

```
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.

## 6.11 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 some examples 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 in fact equal.

In general, it makes sense to define a superclass relationship when the laws for the subclass mention the members of the superclass. For example, it is reasonable to assume, for any pair of `Ord` and `Eq` instances, that if two values are equal under the `Eq` instance, then the `compare` function should return `EQ`. In order 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 in the case where 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 which finds the maximum of a non-empty array of integers. Your function should have type `Partial => Array Int -> Int`. 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 which defines an action of one type on another:

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

   An *action* is a function which describes how monoidal values can be used 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)`

   That is, the action respects the operations defined by the `Monoid` class.

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

   ```
   newtype Multiply = Multiply Int

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

   instance monoidMultiply :: Monoid Multiply where
     mempty = Multiply 1
   ```

   This monoid acts on strings by repeating an input string some number of times. Write an instance which implements this action:

   ```
   instance repeatAction :: Action Multiply String
   ```

   Does this instance satisfy the laws listed above?
3. (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.
4. (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
   ```

5. (Difficult) Should the arguments of the multi-parameter type class `Action` be related by some functional dependency? Why or why not?

## 6.12 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

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 which 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 has been 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 hashInt :: Hashable Int where
  hash = hashCode
```

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

```
instance hashBoolean :: 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 hashChar :: 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 hashArray :: 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 hashString :: 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 which 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 length zero is `[]`. Any two non-empty arrays are equal only if they have equals 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.
2. (Medium) Use the `hashEqual` function to write a function which tests if an array has any duplicate elements, using hash-equality as an approximation to value equality. Remember to check for value equality using == if a duplicate pair is found. *Hint*: the `nubBy` 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 eqHour :: 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`.

## 6.13 Conclusion

In this chapter, we've been introduced to *type classes*, a type-oriented form of abstraction which 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 gave an introduction to the notion of type class laws, a technique for proving properties about code which 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 will be a theme which we will return to throughout the rest of the book.

# 7. Applicative Validation

## 7.1 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.

## 7.2 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 Bower dependencies, many of which we have seen before. There are two new dependencies:

- `purescript-control`, which defines functions for abstracting control flow using type classes like `Applicative`.
- `purescript-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.

## 7.3 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` which 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"
Address { 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 (Address { 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.

## 7.4 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 b c d f. 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 any 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.

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 functorMaybe :: Functor Maybe where
  map f (Just a) = Just (f a)
  map f Nothing  = Nothing

instance applyMaybe :: 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 arbitrary number of arguments.

For functions of one argument, we can just 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`, then 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 (Address { 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.

## 7.5 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 applicativeMaybe :: Applicative Maybe where
  pure x = Just x
```

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

## 7.6 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, and 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.

## 7.7 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
```

Now suppose that this function forms the implementation of a (very simple!) web service with the three arguments provided as query parameters. We want to make sure that the user provided each of the three parameters, so we might use the `Maybe` type to indicate the presence or otherwise 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
```

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`:

```
> :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

> :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.

## 7.8 Combining Effects

As an example of working with applicative functors abstractly, this section will show how to write a function which will generically combine 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 an 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 which 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 was `Just`, otherwise it returns `Nothing`. This is consistent with our intuition of working in a larger language supporting optional values - a list of computations which return 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 which 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. (Easy) Use `lift2` to write lifted versions of the numeric operators `+`, `-`, `*` and `/` which work with optional arguments.
2. (Medium) Convince yourself that the definition of `lift3` given above in terms of `<$>` and `<*>` does type check.
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 which has an optional result.

## 7.9 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 which are 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 be used to 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
Person
  { firstName: "John",
  , lastName: "Smith",
  , address: Address
      { street: "123 Fake St."
      , city: "FakeTown"
      , state: "CA"
      },
  , phones: [ PhoneNumber
                { type: HomePhone
                , number: "555-555-5555"
                }
            , PhoneNumber
                { 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:

```
nonEmpty :: String -> Either String Unit
nonEmpty "" = Left "Field cannot be empty"
nonEmpty _  = Right unit

validatePerson :: Person -> Either String Person
validatePerson (Person o) =
  person <$> (nonEmpty o.firstName *> pure o.firstName)
         <*> (nonEmpty o.lastName  *> pure o.lastName)
         <*> pure o.address
         <*> pure o.phones
```

In the first two lines, we use the `nonEmpty` function to validate a non-empty string. `nonEmpty` returns an error (indicated with the `Left` constructor) if its input is empty, or a successful empty value (`unit`) using the `Right` constructor otherwise. We use the sequencing operator `*>` to indicate that we want to perform two validations, returning the result from the validator on the right. In this case, the validator on the right simply uses `pure` to return the input unchanged.

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 has a limitation which 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 which is provided by the `purescript-validation` library. This functor is called `V`, and it provides the ability to 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 Unit
nonEmpty field "" = invalid ["Field '" <> field <> "' cannot be empty"]
nonEmpty _     _  = pure unit

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

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

`validateAddress` validates an `Address` structure. It checks that the `street` and `city` fields are non-empty, and checks 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.

## 7.10 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 -> R.Regex -> String -> V Errors Unit
matches _     regex value | R.test regex value =
  pure unit
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 (PhoneNumber o) =
  phoneNumber <$> pure o."type"
              <*> (matches "Number" phoneNumberRegex o.number *> pure o.number)
```

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

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

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

## Exercises

1. (Easy) Use a regular expression validator to ensure that the `state` field of the `Address` type contains two alphabetic characters. *Hint*: see the source code for `phoneNumberRegex`.
2. (Medium) Using the `matches` validator, write a validation function which checks that a string is not entirely whitespace. Use it to replace `nonEmpty` where appropriate.

## 7.11 Traversable Functors

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

```
arrayNonEmpty :: forall a. String -> Array a -> V Errors Unit
arrayNonEmpty field [] =
  invalid ["Field '" <> field <> "' must contain at least one value"]
arrayNonEmpty _     _  =
  pure unit

validatePerson :: Person -> V Errors Person
validatePerson (Person o) =
  person <$> (nonEmpty "First Name" o.firstName *>
              pure o.firstName)
         <*> (nonEmpty "Last Name"  o.lastName   *>
              pure o.lastName)
               <*> validateAddress o.address
         <*> (arrayNonEmpty "Phone Numbers" o.phones *>
              traverse validatePhoneNumber o.phones)
```

There is one more interesting function here, which we haven't seen yet - `traverse`, which appears in the final line.

`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 f. Applicative f => (a -> f b) -> t a -> f (t b)
  sequence :: forall a f. Applicative f => t (f a) -> f (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 which supported a fold operation, reducing a structure to a single value). In addition, a traversable functor provides the ability to combine a collection of side-effects which 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 f. Applicative f => (a -> f b) -> Array a -> f (Array b)
```

Intuitively, given any applicative functor `f`, and a function which takes a value of type `a` and returns a value of type `b` (with side-effects tracked by `f`), 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 `f`).

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 `f` for a type `a`, then `traverse f` 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 which 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 f. Applicative f => t (f a) -> f (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 each other. This is left as an exercise for the interested reader.

The `Traversable` instance for lists is given in the `Data.List` module. The definition of `traverse` is given here:

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

In the case of an empty list, we can simply 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 `f` 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

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

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

> traverse (nonEmpty "Example") (Just "Testing")
(Valid (Just unit))
```

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`, and `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. (Medium) Write a `Traversable` instance for the following binary tree data structure, which combines side-effects from left-to-right:

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

   This corresponds to an in-order traversal of the tree. What about a preorder traversal? What about reverse order?
2. (Medium) Modify the code to make the `address` field of the `Person` type optional using `Data.Maybe`. *Hint*: Use `traverse` to validate a field of type `Maybe a`.
3. (Difficult) Try to write `sequence` in terms of `traverse`. Can you write `traverse` in terms of `sequence`?

## 7.12 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. In fact, 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 `purescript-parallel` package provides a type class `Parallel` which supports *parallel computations*. `Parallel` provides a function `parallel` which 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 which can be combined in parallel.

## 7.13 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 which capture some notion of side-effect.
- We saw how applicative functors gave a solution to 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 which provide 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!

# 8. The Eff Monad

## 8.1 Chapter Goals

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

The goal of this chapter is to explain why monads are a useful abstraction, and their connection with *do notation*. We will build upon the address book example of the previous chapters, by using a particular monad to handle the side-effects of building a user interface in the browser. The monad we will use is an important monad in PureScript - the `Eff` monad - used to encapsulate so-called *native* effects.

## 8.2 Project Setup

The source code for this project builds on the source for the previous chapter. The modules from the previous project are included in the `src` directory for this project.

The project adds the following Bower dependencies:

- `purescript-eff`, which defines the `Eff` monad, the subject of the second half of the chapter.
- `purescript-react`, a set of bindings to the React user interface library, which we will use to build a user interface for our address book application.

In addition to the modules from the previous chapter, this chapter's project adds a `Main` module, which provides the entry point to the application, and functions to render the user interface.

To compile this project, first install React using `npm install`, and then build and bundle the JavaScript source with `pulp browserify --to dist/Main.js`. To run the project, open the `html/index.html` file in your web browser.

## 8.3 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` then return the pair `[x, y]`, else fail.

Array comprehensions allow us to write this non-deterministic algorithm in a natural way:

```
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:

```
> 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*.

In general, 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 type 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 by using do notation. Suppose we wanted to read a user's city from a user profile which 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`.

## 8.4 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 that we have 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 bindArray :: 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 bindMaybe :: 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 which 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
```

It is worth noting that 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.

## 8.5 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 expression `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
```

## 8.6 Folding With Monads

As an example of working with monads abstractly, this section will present a function which works with any type constructor in the `Monad` type class. This should serve to 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 that 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
```

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

```
foldl :: forall a b
       . (a -> b -> a)
      -> a
      -> List b
      -> a
```

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. At 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, with the exception of 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 Data.List

> 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.

## 8.7 Monads and Applicatives

Every instance of the `Monad` type class is also an instance of the `Applicative` type class, by virtue of the superclass relationship between the two classes.

However, there is also an implementation of the `Applicative` 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 `Applicative` 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 which 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) Look up the types of the `head` and `tail` functions from the `Data.Array` module in the `purescript-arrays` package. Use do notation with the `Maybe` monad to combine these functions into a function `third` which returns the third element of an array with three or more elements. Your function should return an appropriate `Maybe` type.
2. (Medium) Write a function `sums` 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:

   ```
   > sums []
   [0]

   > sums [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 respectively.
3. (Medium) Confirm that the `ap` function and the `apply` operator agree for the `Maybe` monad.
4. (Medium) Verify that the monad laws hold for the `Monad` instance for the `Maybe` type, as defined in the `purescript-maybe` package.
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)
   ```

   Test your function in PSCi using the `Maybe` and `Array` monads.
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 `Applicative` instance uses the `ap` function defined above. Recall that `lift2` was defined as follows:

   ```
   lift2 :: forall f a b c. Applicative f => (a -> b -> c) -> f a -> f b -> f c
   lift2 f a b = f <$> a <*> b
   ```

## 8.8 Native Effects

We will now look at one particular monad which is of central importance in PureScript - the `Eff` monad.

The `Eff` monad is defined in the Prelude, in the `Control.Monad.Eff` module. It is used to manage so-called *native* side-effects.

What are native side-effects? They are the side-effects which 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 `Eff`. 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 `Eff`. So it is not the effect itself which is native, but rather how it is implemented at runtime.

## 8.9 Side-Effects and Purity

In a pure language like PureScript, one question which presents itself is: without side-effects, how can one write useful real-world code?

The answer is that PureScript does not aim to eliminate side-effects. It aims to represent side-effects 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 not possible to pass a side-effecting argument to a function, for example, and have side-effects performed unexpectedly.

The only way in which side-effects managed by the `Eff` monad will be presented is to run a computation of type `Eff eff a` from JavaScript.

The Pulp 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 `Eff` monad.

In this way, we know exactly what side-effects to expect: exactly those used by `main`. In addition, we can use the `Eff` monad to restrict what types of side-effects `main` is allowed to have, so that we can say with certainty for example, that our application will interact with the console, but nothing else.

## 8.10 The Eff Monad

The goal of the `Eff` monad is to provide a well-typed API for computations with side-effects, while at the same time generating efficient Javascript. It is also called the monad of *extensible effects*, which will be explained shortly.

Here is an example. It uses the `purescript-random` package, which defines functions for generating random numbers:

```
module Main where

import Prelude

import Control.Monad.Eff.Random (random)
import Control.Monad.Eff.Console (logShow)

main = do
  n <- random
  logShow n
```

If this file is saved as `src/Main.purs`, then it can be compiled and run using Pulp:

```
$ pulp run
```

Running this command, you will see a randomly chosen number between `0` and `1` printed to the console.

This program uses do notation to combine two types of native effects provided by the Javascript runtime: random number generation and console IO.

## 8.11 Extensible Effects

We can inspect the type of main by opening the module in PSCi:

```
> import Main

> :type main
forall eff. Eff (console :: CONSOLE, random :: RANDOM | eff) Unit
```

This type looks quite complicated, but is easily explained by analogy with PureScript's records.

Consider a simple function which uses a record type:

```
fullName person = person.firstName <> " " <> person.lastName
```

This function creates a full name string from a record containing `firstName` and `lastName` properties. If you find the type of this function in PSCi as before, you will see this:

```
forall r. { firstName :: String, lastName :: String | r } -> String
```

This type reads as follows: “`fullName` takes a record with `firstName` and `lastName` fields *and any other properties* and returns a `String`”.

That is, `fullName` does not care if you pass a record with more fields, as long as the `firstName` and `lastName` properties are present:

```
> firstName { firstName: "Phil", lastName: "Freeman", location: "Los Angeles" }
Phil Freeman
```

Similarly, the type of `main` above can be interpreted as follows: “`main` is a *computation with side-effects*, which can be run in any environment which supports random number generation and console IO, *and any other types of side effect*, and which returns a value of type `Unit`”.

This is the origin of the name “extensible effects”: we can always extend the set of side-effects, as long as we can support the set of effects that we need.

## 8.12 Interleaving Effects

This extensibility allows code in the `Eff` monad to *interleave* different types of side-effect.

The `random` function which we used has the following type:

```
forall eff1. Eff (random :: RANDOM | eff1) Number
```

The set of effects `(random :: RANDOM | eff1)` here is *not* the same as those appearing in `main`.

However, we can *instantiate* the type of `random` in such a way that the effects do match. If we choose `eff1` to be `(console :: CONSOLE | eff)`, then the two sets of effects become equal, up to reordering.

Similarly, `logShow` has a type which can be specialized to match the effects of `main`:

```
forall eff2. Show a => a -> Eff (console :: CONSOLE | eff2) Unit
```

This time we have to choose `eff2` to be `(random :: RANDOM | eff)`.

The point is that the types of `random` and `logShow` indicate the side-effects which they contain, but in such a way that other side-effects can be *mixed-in*, to build larger computations with larger sets of side-effects.

Note that we don’t have to give a type for `main`. The compiler will find a most general type for `main` given the polymorphic types of `random` and `logShow`.

## 8.13 The Kind of Eff

The type of `main` is unlike other types we’ve seen before. To explain it, we need to consider the *kind* of `Eff`. Recall that types are classified by their kinds just like values are classified by their types. So far, we’ve only seen kinds built from `Type` (the kind of types) and `->` (which builds kinds for type constructors).

To find the kind of `Eff`, use the `:kind` command in PSCi:

```
> import Control.Monad.Eff

> :kind Eff
# Control.Monad.Eff.Effect -> Type -> Type
```

There are two kinds here that we have not seen before.

`Control.Monad.Eff.Effect` is the kind of *effects*, which represents *type-level labels* for different types of side-effects. To understand this, note that the two labels we saw in `main` above both have kind `Control.Monad.Eff.Effect`:

```
> import Control.Monad.Eff.Console
> import Control.Monad.Eff.Random

> :kind CONSOLE
Control.Monad.Eff.Effect

> :kind RANDOM
Control.Monad.Eff.Effect
```

The `#` kind constructor is used to construct kinds for *rows*, i.e. unordered, labelled sets.

So `Eff` is parameterized by a row of effects, and its return type. That is, the first argument to `Eff` is an unordered, labelled set of effect types, and the second argument is the return type.

We can now read the type of `main` above:

```
forall eff. Eff (console :: CONSOLE, random :: RANDOM | eff) Unit
```

The first argument to `Eff` is `(console :: CONSOLE, random :: RANDOM | eff)`. This is a row which contains the `CONSOLE` effect and the `RANDOM` effect. The pipe symbol `|` separates the labelled effects from the *row variable* `eff` which represents *any other side-effects* we might want to mix in.

The second argument to `Eff` is `Unit`, which is the return type of the computation.

## 8.14 Records And Rows

Considering the kind of `Eff` allows us to make a deeper connection between extensible effects and records.

Take the function we defined above:

```
fullName :: forall r. { firstName :: String, lastName :: String | r } -> String
fullName person = person.firstName <> " " <> person.lastName
```

The kind of the type on the left of the function arrow must be `Type`, because only types of kind `Type` have values.

The curly braces are actually syntactic sugar, and the full type as understood by the PureScript compiler is as follows:

```
fullName :: forall r. Record (firstName :: String, lastName :: String | r) -> String
```

Note that the curly braces have been removed, and there is an extra `Record` constructor. `Record` is a built-in type constructor defined in the `Prim` module. If we find its kind, we see the following:

```
> :kind Record
# Type -> Type
```

That is, `Record` is a type constructor which takes a *row of types* and constructs a type. This is what allows us to write row-polymorphic functions on records.

The type system uses the same machinery to handle extensible effects as is used for row-polymorphic records (or *extensible records*). The only difference is the *kind* of the types appearing in the labels. Records are parameterized by a row of types, and `Eff` is parameterized by a row of effects.

The same type system feature could even be used to build other types which were parameterized on rows of type constructors, or even rows of other rows!

## 8.15 Fine-Grained Effects

Type annotations are usually not required when using `Eff`, since rows of effects can be inferred, but they can be used to indicate to the compiler which effects are expected in a computation.

If we annotate the previous example with a *closed* row of effects:

```
main :: Eff (console :: CONSOLE, random :: RANDOM) Unit
main = do
  n <- random
  print n
```

(note the lack of the row variable `eff` here), then we cannot accidentally include a subcomputation which makes use of a different type of effect. In this way, we can control the side-effects that our code is allowed to have.

## 8.16 Handlers and Actions

Functions such as `print` and `random` are called *actions*. Actions have the `Eff` type on the right hand side of their functions, and their purpose is to *introduce* new effects.

This is in contrast to *handlers*, in which the `Eff` type appears as the type of a function argument. While actions *add* to the set of required effects, a handler usually *subtracts* effects from the set.

As an example, consider the `purescript-exceptions` package. It defines two functions, `throwException` and `catchException`:

```
throwException :: forall a eff
                . Error
               -> Eff (exception :: EXCEPTION | eff) a

catchException :: forall a eff
                . (Error -> Eff eff a)
               -> Eff (exception :: EXCEPTION | eff) a
               -> Eff eff a
```

throwException is an action. Eff appears on the right hand side, and introduces the new EXCEPTION effect.

catchException is a handler. Eff appears as the type of the second function argument, and the overall effect is to *remove* the EXCEPTION effect.

This is useful, because the type system can be used to delimit portions of code which require a particular effect. That code can then be wrapped in a handler, allowing it to be embedded inside a block of code which does not allow that effect.

For example, we can write a piece of code which throws exceptions using the Exception effect, then wrap that code using catchException to embed the computation in a piece of code which does not allow exceptions.

Suppose we wanted to read our application's configuration from a JSON document. The process of parsing the document might result in an exception. The process of reading and parsing the configuration could be written as a function with this type signature:

```
readConfig :: forall eff. Eff (exception :: EXCEPTION | eff) Config
```

Then, in the main function, we could use catchException to handle the EXCEPTION effect, logging the error and returning a default configuration:

```
main = do
    config <- catchException printException readConfig
    runApplication config
  where
    printException e = do
      log (message e)
      pure defaultConfig
```

The purescript-eff package also defines the runPure handler, which takes a computation with *no* side-effects, and safely evaluates it as a pure value:

```
type Pure a = Eff () a

runPure :: forall a. Pure a -> a
```

## 8.17 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:

```
newSTRef :: forall a h eff. a -> Eff (st :: ST h | eff) (STRef h a)

readSTRef :: forall a h eff. STRef h a -> Eff (st :: ST h | eff) a

writeSTRef :: forall a h eff. STRef h a -> a -> Eff (st :: ST h | eff) a

modifySTRef :: forall a h eff. STRef h a -> (a -> a) -> Eff (st :: ST h | eff) a
```

`newSTRef` is used to create a new mutable reference cell of type `STRef h a`, which can be read using the `readSTRef` action, and modified using the `writeSTRef` and `modifySTRef` actions. The type `a` is the type of the value stored in the cell, and the type `h` 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 a large number of 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 (using the `forE` action in `Control.Monad.Eff`) to update the value stored in that cell:

```
import Prelude

import Control.Monad.Eff (Eff, forE)
import Control.Monad.ST (ST, newSTRef, readSTRef, modifySTRef)

simulate :: forall eff h. Number -> Number -> Int -> Eff (st :: ST h | eff) Number
simulate x0 v0 time = do
  ref <- newSTRef { x: x0, v: v0 }
  forE 0 (time * 1000) \_ -> do
    modifySTRef ref \o ->
      { v: o.v - 9.81 * 0.001
      , x: o.x + o.v * 0.001
      }
    pure unit
  final <- readSTRef 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 mutable state, it is still a pure function, so long as the reference cell `ref` is not allowed to be used by other parts of the program. 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 `runST` function:

```
runST :: forall a eff. (forall h. Eff (st :: ST h | eff) a) -> Eff eff a
```

The thing to notice here is that the region type `h` is quantified *inside the parentheses* on the left of the function arrow. That means that whatever action we pass to `runST` has to work with *any region* `h` whatsoever.

However, once a reference cell has been created by `newSTRef`, 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 `runST`. This is what allows `runST` to safely remove the `ST` effect!

In fact, since `ST` is the only effect in our example, we can use `runST` in conjunction with `runPure` to turn `simulate` into a pure function:

```
simulate' :: Number -> Number -> Number -> Number
simulate' x0 v0 time = runPure (runST (simulate x0 v0 time))
```

You can even try running this function in PSCi:

```
> import Main

> simulate' 100.0 0.0 0.0
100.00

> simulate' 100.0 0.0 1.0
95.10

> simulate' 100.0 0.0 2.0
80.39

> simulate' 100.0 0.0 3.0
55.87

> simulate' 100.0 0.0 4.0
21.54
```

In fact, if we inline the definition of `simulate` at the call to `runST`, as follows:

```
simulate :: Number -> Number -> Int -> Number
simulate x0 v0 time = runPure $ runST do
  ref <- newSTRef { x: x0, v: v0 }
  forE 0 (time * 1000) \_ -> do
    modifySTRef ref \o ->
      { v: o.v - 9.81 * 0.001
      , x: o.x + o.v * 0.001
      }
    pure unit
  final <- readSTRef ref
  pure final.x
```

then the compiler will notice that the reference cell is not allowed to escape its scope, and can safely turn it into a `var`. Here is the generated JavaScript for the body of the call to `runST`:

```
var ref = { x: x0, v: v0 };

Control_Monad_Eff.forE(0)(time * 1000 | 0)(function (i) {
  return function __do() {
    ref = (function (o) {
      return {
        v: o.v - 9.81 * 1.0e-3,
        x: o.x + o.v * 1.0e-3
      };
    })(ref);
    return Prelude.unit;
  };
})();

return ref.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 `forE`, `foreachE`, `whileE` and `untilE` which generate efficient loops in the `Eff` monad.

## Exercises

1. (Medium) Rewrite the `safeDivide` function to throw an exception using `throwException` if the denominator is zero.
2. (Difficult) The following is a simple way to estimate pi: randomly choose a large number `N` of points in the unit square, and count the number `n` which lie in the inscribed circle. An estimate for pi is `4n/N`. Use the `RANDOM` and `ST` effects with the `forE` function to write a function which estimates pi in this way.

## 8.18 DOM Effects

In the final sections of this chapter, we will apply what we have learned about effects in the `Eff` monad to the problem of working with the DOM.

There are a number of PureScript packages for working directly with the DOM, or with open-source DOM libraries. For example:

- `purescript-dom`[1] is an extensive set of low-level bindings to the browser's DOM APIs.
- `purescript-jquery`[2] is a set of bindings to the jQuery[3] library.

There are also PureScript libraries which build abstractions on top of these libraries, such as

- `purescript-thermite`[4], which builds on `purescript-react`, and
- `purescript-halogen`[5] which provides a type-safe set of abstractions on top of a custom virtual DOM library.

In this chapter, we will use the `purescript-react` library to add a user interface to our address book application, but the interested reader is encouraged to explore alternative approaches.

## 8.19 An Address Book User Interface

Using the `purescript-react` 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 `purescript-react` library uses the `Eff` 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 `Eff` monad.

We are going to build a form which 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 in which 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.

The HTML file is essentially empty, except for the following line:

[1] http://github.com/purescript-contrib/purescript-dom

[2] http://github.com/paf31/purescript-jquery

[3] http://jquery.org

[4] http://github.com/paf31/purescript-thermite

[5] http://github.com/slamdata/purescript-halogen

```
<script type="text/javascript" src="../dist/Main.js"></script>
```

This line includes the JavaScript code which is generated by Pulp. We place it at the end of the file to ensure that the relevant elements are on the page before we try to access them. To rebuild the `Main.js` file, Pulp can be used with the `browserify` command. Make sure the `dist` directory exists first, and that you have installed React as an NPM dependency:

```
$ npm install # Install React
$ mkdir dist/
$ pulp browserify --to dist/Main.js
```

The `Main` defines the `main` function, which creates the address book component, and renders it to the screen. The `main` function uses the `CONSOLE` and `DOM` effects only, as its type signature indicates:

```
main :: Eff (console :: CONSOLE, dom :: DOM) Unit
```

First, `main` logs a status message to the console:

```
main = void do
  log "Rendering address book component"
```

Later, `main` uses the DOM API to obtain a reference (`doc`) to the document body:

```
  doc <- window >>= document
```

Note that this provides an example of interleaving effects: the `log` function uses the `CONSOLE` effect, and the `window` and `document` functions both use the `DOM` effect. The type of `main` indicates that it uses both effects.

`main` uses the `window` action to get a reference to the window object, and passes the result to the `document` function using `>>=`. `document` takes a window object and returns a reference to its document.

Note that, by the definition of do notation, we could have instead written this as follows:

```
  w <- window
  doc <- document w
```

It is a matter of personal preference whether this is more or less readable. The first version is an example of *point-free* form, since there are no function arguments named, unlike the second version which uses the name `w` for the window object.

The `Main` module defines an address book *component*, called `addressBook`. To understand its definition, we will need to first need to understand some concepts.

In order to create a React component, we must first create a React *class*, which acts like a template for a component. In `purescript-react`, we can create classes using the `createClass` function. `createClass` requires a *specification* of our class, which is essentially a collection of `Eff` actions which are used to handle various parts of the component's lifecycle. The action we will be interested in is the `Render` action.

Here are the types of some relevant functions provided by the React library:

```
createClass
  :: forall props state eff
   . ReactSpec props state eff
  -> ReactClass props

type Render props state eff
   = ReactThis props state
  -> Eff ( props :: ReactProps
         , refs :: ReactRefs Disallowed
         , state :: ReactState ReadOnly
         | eff
         ) ReactElement

spec
  :: forall props state eff
   . state
  -> Render props state eff
  -> ReactSpec props state eff
```

There are a few interesting things to note here:

- The `Render` type synonym is provided in order to simplify some type signatures, and it represents the rendering function for a component.
- A `Render` action takes a reference to the component (of type `ReactThis`), and returns a `ReactElement` in the `Eff` monad. A `ReactElement` is a data structure describing our intended state of the DOM after rendering.
- Every React component defines some type of state. The state can be changed in response to events like button clicks. In `purescript-react`, the initial state value is provided in the `spec` function.
- The effect row in the `Render` type uses some interesting effects to restrict access to the React component's state in certain functions. For example, during rendering, access to the "refs" object is `Disallowed`, and access to the component state is `ReadOnly`.

The `Main` module defines a type of states for the address book component, and an initial state:

```
newtype AppState = AppState
  { person :: Person
  , errors :: Errors
  }

initialState :: AppState
initialState = AppState
  { person: examplePerson
  , errors: []
  }
```

The state contains a Person record (which we will make editable using form components), and a collection of errors (which will be populated using our existing validation code).

Now let's see the definition of our component:

```
addressBook :: forall props. ReactClass props
```

As already indicated, addressBook will use createClass and spec to create a React class. To do so, it will provide our initial state value, and a Render action. However, what can we do in the Render action? To answer that, purescript-react provides some simple actions which can be used:

```
readState
  :: forall props state access eff
   . ReactThis props state
  -> Eff ( state :: ReactState ( read :: Read
                               | access
                               )
         | eff
         ) state

writeState
  :: forall props state access eff
   . ReactThis props state
  -> state
  -> Eff ( state :: ReactState ( write :: Write
                               | access
                               )
         | eff
         ) state
```

The readState and writeState functions use extensible effects to ensure that we have access to the React state (via the ReactState effect), but note that read and write permissions are separated further, by parameterizing the ReactState effect on *another* row!

This illustrates an interesting point about PureScript's row-based effects: effects appearing inside rows need not be simple singletons, but can have interesting structure, and this flexibility enables some useful restrictions at compile time. If the purescript-react library did not make this restriction then it would be possible to get exceptions at runtime if we tried to write the state in the Render action, for example. Instead, such mistakes are now caught at compile time.

Now we can read the definition of our addressBook component. It starts by reading the current component state:

```
addressBook = createClass $ spec initialState \ctx -> do
  AppState { person: Person person@{ homeAddress: Address address }
           , errors
           } <- readState ctx
```

Note the following:

- The name `ctx` refers to the `ReactThis` reference, and can be used to read and write the state where appropriate.
- The record inside `AppState` is matched using a record binder, including a record pun for the *errors* field. We explicitly name various parts of the state structure for convenience.

Recall that `Render` must return a `ReactElement` structure, representing the intended state of the DOM. The `Render` action is defined in terms of some helper functions. One such helper function is `renderValidationErrors`, which turns the `Errors` structure into an array of `ReactElements`.

```
renderValidationError :: String -> ReactElement
renderValidationError err = D.li' [ D.text err ]

renderValidationErrors :: Errors -> Array ReactElement
renderValidationErrors [] = []
renderValidationErrors xs =
  [ D.div [ P.className "alert alert-danger" ]
          [ D.ul' (map renderValidationError xs) ]
  ]
```

In `purescript-react`, `ReactElements` are typically created by applying functions like `div`, which create single HTML elements. These functions usually take an array of attributes, and an array of child elements as arguments. However, names ending with a prime character (like `ul'` here) omit the attribute array, and use the default attributes instead.

Note that since we are simply manipulating regular data structures here, we can use functions like `map` to build up more interesting elements.

A second helper function is `formField`, which creates a `ReactElement` containing a text input for a single form field:

```
formField
  :: String
  -> String
  -> String
  -> (String -> Person)
  -> ReactElement
formField name hint value update =
  D.div [ P.className "form-group" ]
        [ D.label [ P.className "col-sm-2 control-label" ]
                  [ D.text name ]
        , D.div [ P.className "col-sm-3" ]
                [ D.input [ P._type "text"
                          , P.className "form-control"
                          , P.placeholder hint
                          , P.value value
                          , P.onChange (updateAppState ctx update)
                          ] []
                ]
        ]
```

Again, note that we are composing more interesting elements from simpler elements, applying attributes to each element as we go. One attribute of note here is the `onChange` attribute applied to the `input` element. This is an *event handler*, and is used to update the component state when the user edits text in our text box. Our event handler is defined using a third helper function, `updateAppState`:

```
updateAppState
  :: forall props eff
   . ReactThis props AppState
  -> (String -> Person)
  -> Event
  -> Eff ( console :: CONSOLE
         , state :: ReactState ReadWrite
         | eff
         ) Unit
```

`updateAppState` takes a reference to the component in the form of our `ReactThis` value, a function to update the `Person` record, and the `Event` record we are responding to. First, it extracts the new value of the text box from the `change` event (using the `valueOf` helper function), and uses it to create a new `Person` state:

```
  for_ (valueOf e) \s -> do
    let newPerson = update s
```

Then, it runs the validation function, and updates the component state (using `writeState`) accordingly:

```
  log "Running validators"
  case validatePerson' newPerson of
    Left errors ->
      writeState ctx (AppState { person: newPerson
                               , errors: errors
                               })
    Right _ ->
      writeState ctx (AppState { person: newPerson
                               , errors: []
                               })
```

That covers the basics of our component implementation. However, you should read the source accompanying this chapter in order to get a full understanding of the way the component works.

Also try the user interface out by running `pulp browserify --to dist/Main.js` and then opening the `html/index.html` file in your web browser. You should be able to enter some values into the form fields and see the validation errors printed onto the page.

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

1. (Easy) Modify the application to include a work phone number text box.
2. (Medium) Instead of using a `ul` element to show the validation errors in a list, modify the code to create one `div` with the `alert` style 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 which extracts the validation error for a particular `Field` from the `Errors` structure.

## 8.20 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 which 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 `Eff` monad, which is used to handle native side-effects.
- We saw how the `Eff` monad supports extensible effects, and how multiple types of native effect can be interleaved into the same computation.
- We saw how effects and records are handled in the kind system, and the connection between extensible records and extensible effects.
- We used the `Eff` monad to handle a variety of effects: random number generation, exceptions, console IO, mutable state, and DOM manipulation using React.

The `Eff` 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.

# 9. Canvas Graphics

## 9.1 Chapter Goals

This chapter will be an extended example focussing on the `purescript-canvas` package, which provides a way to generate 2D graphics from PureScript using the HTML5 Canvas API.

## 9.2 Project Setup

This module's project introduces the following new Bower dependencies:

- `purescript-canvas`, which gives types to methods from the HTML5 Canvas API
- `purescript-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 Pulp 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.

## 9.3 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 `Control.Monad.Eff` module, and also the `Graphics.Canvas` module, which contains actions in the `Eff` 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:

```
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 :: forall eff. String -> Eff (canvas :: CANVAS | eff) (Maybe CanvasEl\
ement)

getContext2D :: forall eff. CanvasElement -> Eff (canvas :: CANVAS | eff) 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, by using the setFillStyle action:

```
  setFillStyle "#0000FF" ctx
```

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 eff a. Context2D ->
                          Eff (canvas :: CANVAS | eff) a ->
                          Eff (canvas :: CANVAS | eff) a
```

fillPath takes a graphics context, and another action which builds the path to render. To build a path, we can use the rect action. rect takes a graphics context, and a record which provides the position and size of the rectangle:

```
  fillPath ctx $ rect ctx
    { x: 250.0
    , y: 250.0
    , w: 100.0
    , h: 100.0
    }
```

Build the rectangle example, providing Example.Rectangle as the name of the main module:

```
$ mkdir dist/
$ pulp build -O --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.

## 9.4 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 be used to 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
  , w :: Number
  , h :: Number
  }
```

The `x` and `y` properties represent the location of the top-left corner, while the `w` and `h` properties represent the width and height 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
  , r     :: Number
  , start :: Number
  , end   :: Number
  }
```

Here, the `x` and `y` properties represent the center point, `r` is the radius, and `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`:

```
  fillPath ctx $ arc ctx
    { x      : 300.0
    , y      : 300.0
    , r      : 50.0
    , start  : Math.pi * 5.0 / 8.0
    , end    : Math.pi * 2.0
    }
```

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 that we can write row-polymorphic functions which can act on either type of record.

For example, the `Shapes` module defines a `translate` function which 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 can be 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 "#FF0000" ctx

  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:

```
$ pulp build -O --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 of the examples so far.
2. (Easy) The `fillPath` and `strokePath` functions can be used to render complex paths with a common style by 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
     :: forall eff
      . Context2D
     -> Array Point
     -> Eff (canvas :: Canvas | eff) 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 which 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`.

## 9.5 Drawing Random Circles

The `Example/Random.purs` file contains an example which uses the `Eff` monad to interleave two different 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 "#FF0000" ctx
  setStrokeStyle "#000000" ctx
```

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
         , r     : r * 50.0
         , start : 0.0
         , end   : Math.pi * 2.0
         }
    fillPath ctx path
    strokePath ctx path
```

Build this example by specifying the `Example.Random` module as the main module:

```
$ pulp build -O --main Example.Random --to dist/Main.js
```

and view the result by opening `html/index.html`.

## 9.6 Transformations

There is more to the canvas than just rendering simple shapes. Every canvas maintains a transformation which is used to transform shapes before rendering. Shapes can be translated, rotated, scaled, and skewed.

The `purescript-canvas` library supports these transformations using the following functions:

```
translate :: forall eff
           . TranslateTransform
          -> Context2D
          -> Eff (canvas :: Canvas | eff) Context2D

rotate    :: forall eff
           . Number
          -> Context2D
          -> Eff (canvas :: Canvas | eff) Context2D

scale     :: forall eff
           . ScaleTransform
          -> Context2D
          -> Eff (canvas :: CANVAS | eff) Context2D

transform :: forall eff
           . Transform
          -> Context2D
          -> Eff (canvas :: CANVAS | eff) Context2D
```

The `translate` action performs a translation whose components are specified by the properties of the `TranslateTransform` record.

The `rotate` action performs a rotation 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 { translateX: 10.0, translateY: 10.0 } ctx
  scale { scaleX: 2.0, scaleY: 2.0 } ctx
  rotate (Math.pi / 2.0) ctx

  renderScene
```

The effect of this sequence of actions is that the scene is rotated, then scaled, and finally translated.

## 9.7 Preserving the Context

A common use case is to render some subset of the scene using a transformation, and then to reset the transformation afterwards.

The Canvas API provides the `save` and `restore` methods, which manipulate a *stack* of states associated with the canvas. `purescript-canvas` wraps this functionality into the following functions:

```
save
  :: forall eff
   . Context2D
  -> Eff (canvas :: CANVAS | eff) Context2D

restore
  :: forall eff
   . Context2D
  -> Eff (canvas :: CANVAS | eff) 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.pi ctx
  render
  restore ctx
```

In the interest of abstracting over common use cases using higher-order functions, the `purescript-canvas` library provides the `withContext` function, which performs some canvas action while preserving the original context state:

```
withContext
  :: forall eff a
   . Context2D
  -> Eff (canvas :: CANVAS | eff) a
  -> Eff (canvas :: CANVAS | eff) a
```

We could rewrite the `rotated` function above using `withContext` as follows:

```
rotated ctx render =
  withContext ctx do
    rotate Math.pi ctx
    render
```

## 9.8 Global Mutable State

In this section, we'll use the `purescript-refs` package to demonstrate another effect in the `Eff` monad.

The `Control.Monad.Eff.Ref` module provides a type constructor for global mutable references, and an associated effect:

```
> import Control.Monad.Eff.Ref

> :kind Ref
Type -> Type

> :kind REF
Control.Monad.Eff.Effect
```

A value of type `Ref a` is a mutable reference cell containing a value of type `a`, much like an `STRef h a`, which we saw in the previous chapter. The difference is that, while the `ST` effect can be removed by using `runST`, the `Ref` effect does not provide a handler. Where `ST` is used to track safe, local mutation, `Ref` is used to track global mutation. As such, it should be used sparingly.

The `Example/Refs.purs` file contains an example which uses the `REF` effect to track mouse clicks on the `canvas` element.

The code starts by creating a new reference containing the value `0`, by using the `newRef` action:

```
  clickCount <- newRef 0
```

Inside the click event handler, the `modifyRef` action is used to update the click count:

```
    modifyRef clickCount (\count -> count + 1)
```

The `readRef` action is used to read the new click count:

```
    count <- readRef clickCount
```

In the `render` function, the click count is used to determine the transformation applied to a rectangle:

```
withContext ctx do
  let scaleX = Math.sin (toNumber count * Math.pi / 4.0) + 1.5
  let scaleY = Math.sin (toNumber count * Math.pi / 6.0) + 1.5

  translate { translateX: 300.0, translateY:  300.0 } ctx
  rotate (toNumber count * Math.pi / 18.0) ctx
  scale { scaleX: scaleX, scaleY: scaleY } ctx
  translate { translateX: -100.0, translateY: -100.0 } ctx

  fillPath ctx $ rect ctx
    { x: 0.0
    , y: 0.0
    , w: 200.0
    , h: 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 that 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 that it center lies at the center of the canvas.

Build the example:

```
$ pulp build -O --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.

## Examples

1. (Easy) Write a higher-order function which strokes and fills a path simultaneously. Rewrite the `Random.purs` example using your function.
2. (Medium) Use the `RANDOM` and `DOM` effects to create an application which renders a circle with random position, color and radius to the canvas when the mouse is clicked.
3. (Medium) Write a function which transforms the scene by rotating it around a point with specified coordinates. *Hint*: use a translation to first translate the scene to the origin.

## 9.9 L-Systems

In this final example, we will use the `purescript-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 instruction approximates a curve called 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 choice of alphabet by a choice of type. For our example, we can choose the following type:

```
data Alphabet = 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 Alphabet

initial :: Sentence
initial = [F, R, R, F, R, R, F, R, R]
```

Our production rules can be represented as a function from `Alphabet` to `Sentence` as follows:

```
productions :: Alphabet -> 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` which 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 which can render a letter of the alphabet to the canvas.

Here is a first approximation to the type of `lsystem`:

```
forall eff. Sentence
         -> (Alphabet -> Sentence)
         -> (Alphabet -> Eff (canvas :: Canvas | eff) Unit)
         -> Int
         -> Eff (canvas :: CANVAS | eff) Unit
```

The first two argument types correspond to the values `initial` and `productions`.

The third argument represents a function which 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 `Alphabet`, but for any type, so we should generalize our type accordingly. Let's replace `Alphabet` and `Sentence` with `a` and `Array a` for some quantified type variable `a`:

```
forall a eff. Array a
           -> (a -> Array a)
           -> (a -> Eff (canvas :: CANVAS | eff) Unit)
           -> Int
           -> Eff (canvas :: CANVAS | eff) Unit
```

The second observation is that, in order 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 eff. Array a
             -> (a -> Array a)
             -> (s -> a -> Eff (canvas :: CANVAS | eff) s)
             -> Int
             -> s
             -> Eff (canvas :: CANVAS | eff) 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 eff
         . Array a
        -> (a -> Array a)
        -> (s -> a -> Eff (canvas :: CANVAS | eff) s)
        -> Int
        -> s
        -> Eff (canvas :: CANVAS | eff) 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 when n is non-zero.

In the first case, the recursion is complete, and we simply 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 -> Eff (canvas :: CANVAS | eff) s. This sounds like a job for the foldM function which we defined earlier, and which is available from the purescript-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 n = go (concatMap prod s) (n - 1)
```

That's it! Note how the use of 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 Eff 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, or print information to the console, or support failure or multiple return values. The reader is encouraged to try writing L-systems which 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 gain the freedom to interpret our data in multiple different 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 Alphabet and s to be State, and for the monad m we can choose Eff (canvas :: CANVAS). This gives us the following type:

```
interpret :: State -> Alphabet -> Eff (canvas :: CANVAS) State
```

To implement this function, we need to handle the three data constructors of the `Alphabet` 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 - Math.pi / 3 }
interpret state R = pure $ state { theta = state.theta + Math.pi / 3 }
```

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 + Math.cos state.theta * 1.5
      y = state.y + Math.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

```
$ pulp build -O --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, by 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 (`pi/3`). Instead, it can be moved into the `Alphabet` data type, which allows it to be changed by the production rules:

   ```
   type Angle = Number

   data Alphabet = 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 Alphabet = 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 the `CONSOLE` effect to write the L-system onto the console, or using the `RANDOM` effect to apply random "mutations" to the state type.

## 9.10 Conclusion

In this chapter, we learned how to use the HTML5 Canvas API from PureScript by using the `purescript-canvas` library. We also saw a practical demonstration of many of the techniques we have learned already: maps and folds, records and row polymorphism, and the `Eff` 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 `purescript-drawing` package, and it brings the flexibility of being able to manipulate the scene as data in various ways before rendering.

In the next chapter, we will see how to implement libraries like `purescript-canvas` which wrap existing JavaScript functionality, by using PureScript's *foreign function interface*.

# 10. The Foreign Function Interface

## 10.1 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 the following:

- How to call pure JavaScript functions from PureScript,
- How to create new effect types and actions for use with the `Eff` monad, based on existing JavaScript code,
- How to call PureScript code from JavaScript,
- How to understand the representation of PureScript values at runtime,
- How to work with untyped data using the `purescript-foreign` 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:

- Alerting the user with a popup notification,
- Storing the serialized form data in the browser's local storage, and reloading it when the application restarts.

## 10.2 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 adds two new Bower dependencies:

1. The `purescript-foreign` library, which provides a data type and functions for working with *untyped data*.
2. The `purescript-foreign-generic` library, which adds support for *datatype generic programming* to the `purescript-foreign` library.

*Note*: to avoid browser-specific issues with local storage when the webpage is served from a local file, it might be necessary to run this chapter's project over HTTP.

## 10.3 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 have an understanding of 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.

## 10.4 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:

```
var Test = require('Test');
Test.gcd(15)(20);
```

Here, I am assuming that the code was compiled with `pulp build`, which compiles PureScript modules to CommonJS modules. For that reason, I was able to reference the `gcd` function on the `Test` object, after importing the `Test` module using `require`.

You might also like to bundle JavaScript code for the browser, using `pulp build -O --to file.js`. In that case, you would access the `Test` module from the global PureScript namespace, which defaults to `PS`:

```
var Test = PS.Test;
Test.gcd(15)(20);
```

## 10.5 Understanding Name Generation

PureScript aims to preserve names during code generation as much as possible. In particular, most identifiers which 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.

## 10.6 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 not be distinguished from values of type Number by using typeof.

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 which 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. Just 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.

## 10.7 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 which 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 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 actually the same as the type of its argument.

For example, this newtype representing telephone numbers:

```
newtype PhoneNumber = PhoneNumber String
```

is actually represented as a JavaScript string at runtime. This is useful for designing libraries, since newtypes provide an additional layer of type safety, but without the runtime overhead of another function call.

## 10.8 Representing Quantified Types

Expressions with quantified (polymorphic) types have restrictive representations at runtime. In practice, this means that there are relatively few expressions with a given quantified type, but that 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 - namely, the identity function `id`, defined in the `Prelude`:

```
id :: forall a. a -> a
id a = a
```

In fact, the `id` function is the *only* (total) function with this type! This certainly seems to be the case (try writing an expression with this type which is not observably equivalent to `id`), 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, and so `id` 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), and so this representation of polymorphic data is appropriate.

## 10.9 Representing Constrained Types

Functions with a type class constraint have an interesting representation at runtime. Because the behavior of the function 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 which 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 the Prelude as the first parameter:

```
shout(require('Prelude').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 `purescript-arrays` package, calling them from JavaScript, by compiling the library using `pulp build` and importing modules using the `require` function in NodeJS. *Hint*: you may need to configure the output path so that the generated CommonJS modules are available on the NodeJS module path.

## 10.10 Using JavaScript Code 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 should have a corresponding Javascript declaration in a *foreign Javascript module*.

For example, consider the `encodeURIComponent` function, which can be used from 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 Data.URI where

foreign import encodeURIComponent :: String -> String
```

We also need to write a foreign Javascript module. If the module above is saved as `src/Data/URI.purs`, then the foreign Javascript module should be saved as `src/Data/URI.js`:

```
"use strict";

exports.encodeURIComponent = encodeURIComponent;
```

Pulp will find `.js` files in the `src` directory, and provide them to the compiler as foreign Javascript modules.

Javascript functions and values are exported from foreign Javascript modules by assigning them to the `exports` object just like a regular CommonJS module. The `purs` compiler treats this module like a regular CommonJS module, and simply adds it as a dependency to the compiled PureScript module. However, when bundling code for the browser with `psc-bundle` or `pulp build -O --to`, it is very important to follow the pattern above, assigning exports to the `exports` object using a property assignment. This is because `psc-bundle` recognizes this format, allowing unused Javascript exports to be removed from bundled code.

With these two pieces in place, we can now use the `encodeURIComponent` function from PureScript like any function written in PureScript. For example, if this declaration is saved as a module and loaded into PSCi, we can reproduce the calculation above:

```
$ pulp repl

> import Data.URI
> encodeURIComponent "Hello World"
"Hello%20World"
```

This approach works well for simple JavaScript values, but is of limited use for more complicated examples. The reason is that most idiomatic JavaScript code does not meet the strict criteria imposed by the runtime representations of the basic PureScript types. In those cases, we have another option - we can *wrap* the JavaScript code in such a way that we can force it to adhere to the correct runtime representation.

## 10.11 Wrapping JavaScript Values

We might want to wrap Javascript values and functions for a number of reasons:

- A function takes multiple arguments, but we want to call it like a curried function.
- We might want to use the `Eff` monad to keep track of any JavaScript side-effects.
- It might be necessary to handle corner cases like `null` or `undefined`, to give a function the correct runtime representation.

For example, suppose we wanted to recreate the `head` function on arrays by using a foreign declaration. In JavaScript, we might write the function as follows:

```
function head(arr) {
    return arr[0];
}
```

However, there is a problem with this function. We might try to give it the type `forall a. Array a -> a`, but for empty arrays, this function returns `undefined`. Therefore, this function does not have the correct runtime representation, and we should use a wrapper function to handle this corner case.

To keep things simple, we can throw an exception in the case of an empty array. Strictly speaking, pure functions should not throw exceptions, but it will suffice for demonstration purposes, and we can 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:

```
exports.unsafeHead = function(arr) {
  if (arr.length) {
    return arr[0];
  } else {
    throw new Error('unsafeHead: empty array');
  }
};
```

## 10.12 Defining Foreign Types

Throwing an exception in the case of failure is less than ideal - idiomatic PureScript code uses the type system to represent side-effects such as missing values. One example of this approach is the `Maybe` type constructor. In this section, we will build another solution using the FFI.

Suppose we wanted to define a new type `Undefined a` whose representation at runtime was like that for the type `a`, but also allowing the `undefined` value.

We can define a *foreign type* using the FFI using a *foreign type declaration*. The syntax is similar to defining a foreign function:

```
foreign import data Undefined :: Type -> Type
```

Note that 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 simplify our original definition for `head`:

```
exports.head = function(arr) {
  return arr[0];
};
```

And in the PureScript module:

```
foreign import head :: forall a. Array a -> Undefined a
```

Note the two changes: the body of the head function is now much simpler, and returns arr[0] even if that value is undefined, and the type signature has been changed to reflect the fact that our function can return an undefined value.

This function has the correct runtime representation for its type, but is quite useless since we have no way to use a value of type Undefined a. But we can fix that by writing some new functions using the FFI!

The most basic function we need will tell us whether a value is defined or not:

```
foreign import isUndefined :: forall a. Undefined a -> Boolean
```

This is easily defined in our foreign Javascript module as follows:

```
exports.isUndefined = function(value) {
  return value === undefined;
};
```

We can now use isUndefined and head together from PureScript to define a useful function:

```
isEmpty :: forall a. Array a -> Boolean
isEmpty = isUndefined <<< head
```

Here, the foreign functions we defined were very simple, which meant that we were able to benefit from the use of 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.

## 10.13 Functions of Multiple Arguments

PureScript's Prelude contains an interesting set of examples of foreign types. As we have covered already, PureScript's function types only take a single argument, and can be used to simulate functions of multiple arguments via *currying*. This has certain advantages - we can partially apply functions, and give type class instances for function types - but it comes with a performance penalty. For performance critical code, it is sometimes necessary to define genuine JavaScript functions which accept multiple arguments. The Prelude defines foreign types which allow us to work safely with such functions.

For example, the following foreign type declaration is taken from the Prelude in the Data.Function.Uncurried module:

```
foreign import data Fn2 :: Type -> Type -> Type -> Type
```

This defines the type constructor Fn2 which takes three type arguments. Fn2 a b c is a type representing JavaScript functions of two arguments of types a and b, and with return type c.

The purescript-functions package defines similar type constructors for function arities from 0 to 10.

We can create a function of two arguments by using the mkFn2 function, as follows:

```
import Data.Function.Uncurried

divides :: Fn2 Int Int Boolean
divides = mkFn2 \n m -> m % n == 0
```

and we can apply a function of two arguments by using the `runFn2` function:

```
> runFn2 divides 2 10
true

> runFn2 divides 3 10
false
```

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:

```
exports.divides = function(n, m) {
    return m % n === 0;
};
```

## 10.14 Representing Side Effects

The `Eff` monad is also defined as a foreign type in the Prelude. Its runtime representation is quite simple - an expression of type `Eff eff 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 `Eff` type constructor is given in the `Control.Monad.Eff` module as follows:

```
foreign import data Eff :: # Effect -> Type -> Type
```

Recall that the `Eff` type constructor is parameterized by a row of effects and a return type, which is reflected in its kind.

As a simple example, consider the `random` function defined in the `purescript-random` package. Recall that its type was:

```
foreign import random :: forall eff. Eff (random :: RANDOM | eff) Number
```

The definition of the `random` function is given here:

```
exports.random = function() {
  return 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, and 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 `Control.Monad.Eff.Console` module in the `purescript-console` package. The `log` function has the following type:

```
foreign import log :: forall eff. String -> Eff (console :: CONSOLE | eff) Unit
```

And here is its definition:

```
exports.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.

The effects `RANDOM` and `CONSOLE` are also defined as foreign types. Their kinds are defined to be `Effect`, the kind of effects. For example:

```
foreign import data RANDOM :: Effect
```

In fact, it is possible to define new effects in this way, as we will soon see.

Expressions of type `Eff eff a` can be invoked from JavaScript like regular JavaScript methods. For example, since the `main` function is required to have type `Eff eff a` for some set of effects `eff` and some type `a`, it can be invoked as follows:

```
require('Main').main();
```

When using `pulp build -O --to` or `pulp run`, this call to `main` is generated automatically, whenever the `Main` module is defined.

## 10.15 Defining New Effects

The source code for this chapter defines two new effects. The simplest is the `ALERT` effect, defined in the `Control.Monad.Eff.Alert` module. It is used to indicate that a computation might alert the user using a popup window.

The effect is defined first, using a foreign type declaration:

```
foreign import data ALERT :: Effect
```

ALERT is given the kind Effect, indicating that it represents an effect, as opposed to a type.

Next, the alert action is defined. The alert action displays a popup, and adds the ALERT effect to the row of effects:

```
foreign import alert :: forall eff. String -> Eff (alert :: ALERT | eff) Unit
```

The foreign Javascript module is straightforward, defining the alert function by assigning it to the exports variable:

```
"use strict";

exports.alert = function(msg) {
    return function() {
        window.alert(msg);
    };
};
```

The alert action is very similar to the log action from the Control.Monad.Eff.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.

Note that, as in the case of log, the alert function uses a function of no arguments to represent the computation of type Eff (alert :: ALERT | eff) Unit.

The second effect defined in this chapter is the STORAGE effect, which is defined in the Control.Monad.Eff.Storage module. It is used to indicate that a computation might read or write values using the Web Storage API.

The effect is defined in the same way:

```
foreign import data STORAGE :: Effect
```

The Control.Monad.Eff.Storage module defines two actions: getItem, which retrieves a value from local storage, and setItem which inserts or updates a value in local storage. The two functions have the following types:

```
foreign import getItem
  :: forall eff
   . String
  -> Eff (storage :: STORAGE | eff) Foreign

foreign import setItem
  :: forall eff
   . String
  -> String
  -> Eff (storage :: STORAGE | eff) Unit
```

The interested reader can inspect the source code for this module to see the definitions of these actions.

`setItem` takes a key and a value (both strings), and returns a computation which stores the value in local storage at the specified key.

The type of `getItem` is more interesting. It 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 `Foreign` which is defined by the `purescript-foreign` package in the `Data.Foreign` module.

`Data.Foreign` provides a way to work with *untyped data*, or more generally, data whose runtime representation is uncertain.

### Exercises

1. (Medium) Write a wrapper for the `confirm` method on the JavaScript `Window` object, and add your foreign function to the `Control.Monad.Eff.Alert` module.
2. (Medium) Write a wrapper for the `removeItem` method on the `localStorage` object, and add your foreign function to the `Control.Monad.Eff.Storage` module.

## 10.16 Working With Untyped Data

In this section, we will see how we can use the `Data.Foreign` library to turn untyped data into typed data, with the correct runtime representation for its type.

The code for this chapter builds on the address book example from chapter 8, by adding a Save button at the bottom of the form. When the Save button is clicked, the state of the form is serialized to JSON and stored in local storage. When the page is reloaded, the JSON document is retrieved from local storage and parsed.

The `Main` module defines a type for the saved form data:

```
newtype FormData = FormData
  { firstName  :: String
  , lastName   :: String
  , street     :: String
  , city       :: String
  , state      :: String
  , homePhone  :: String
  , cellPhone  :: String
  }
```

The problem is that we have no guarantee that the JSON will have the correct form. Put another way, we don't know that the JSON represents the correct type of data at runtime. This is the sort of problem that is solved by the `purescript-foreign` library. Here are some other examples:

- A JSON response from a web service
- A value passed to a function from JavaScript code

Let's try the `purescript-foreign` and `purescript-foreign-generic` libraries in PSCi.

Start by importing some modules:

```
> import Data.Foreign
> import Data.Foreign.Generic
> import Data.Foreign.JSON
```

A good way to obtain a `Foreign` value is to parse a JSON document. `purescript-foreign-generic` defines the following two functions:

```
parseJSON :: String -> F Foreign
decodeJSON :: forall a. Decode a => String -> F a
```

The type constructor `F` is actually just a type synonym, defined in `Data.Foreign`:

```
type F = Except (NonEmptyList ForeignError)
```

Here, `Except` is an monad for handling exceptions in pure code, much like `Either`. We can convert a value in the `F` monad into a value in the `Either` monad by using the `runExcept` function.

Most of the functions in the `purescript-foreign` and `purescript-foreign-generic` libraries return a value in the `F` monad, which means that we can use do notation and the applicative functor combinators to build typed values.

The `Decode` type class represents those types which can be obtained from untyped data. There are type class instances defined for the primitive types and arrays, and we can define our own instances as well.

Let's try parsing some simple JSON documents using `decodeJSON` in PSCi (remembering to use `runExcept` to unwrap the results):

```
> import Control.Monad.Except

> runExcept (decodeJSON "\"Testing\"" :: F String)
Right "Testing"

> runExcept (decodeJSON "true" :: F Boolean)
Right true

> runExcept (decodeJSON "[1, 2, 3]" :: F (Array Int))
Right [1, 2, 3]
```

Recall that in the `Either` monad, the `Right` data constructor indicates success. Note however, that invalid JSON, or an incorrect type leads to an error:

```
> runExcept (decodeJSON "[1, 2, true]" :: F (Array Int))
(Left (NonEmptyList (NonEmpty (ErrorAtIndex 2 (TypeMismatch "Int" "Boolean")) Nil)))
```

The `purescript-foreign-generic` library tells us where in the JSON document the type error occurred.

## 10.17 Handling Null and Undefined Values

Real-world JSON documents contain null and undefined values, so we need to be able to handle those too.

`purescript-foreign-generic` defines a type constructors which solves this problem: `NullOrUndefined`. It serves a similar purpose to the `Undefined` type constructor that we defined earlier, but uses the `Maybe` type constructor internally to represent missing values.

The module also provides a function `unNullOrUndefined` to unwrap the inner value. We can lift the appropriate function over the `decodeJSON` action to parse JSON documents which permit null values:

```
> import Prelude
> import Data.Foreign.NullOrUndefined

> runExcept (unNullOrUndefined <$> decodeJSON "42" :: F (NullOrUndefined Int))
(Right (Just 42))

> runExcept (unNullOrUndefined <$> decodeJSON "null" :: F (NullOrUndefined Int))
(Right Nothing)
```

In each case, the type annotation applies to the term to the right of the `<$>` operator. For example, `decodeJSON "42"` has the type `F (NullOrUndefined Int)`. The `unNullOrUndefined` function is then lifted over `F` to give the final type `F (Maybe Int)`.

The type `NullOrUndefined Int` represents values which are either integers, or null. What if we wanted to parse more interesting values, like arrays of integers, where each element might be `null`? In that case, we could lift the function `map unNullOrUndefined` over the `decodeJSON` action, as follows:

```
> runExcept (map unNullOrUndefined <$> decodeJSON "[1, 2, null]" :: F (Array (NullOrUndefi\
ned Int)))
(Right [(Just 1),(Just 2),Nothing])
```

In general, using newtypes to wrap an existing type is a good way to provide different serialization strategies for the same type. The `NullOrUndefined` type is defined as a newtype around the `Maybe` type constructor.

## 10.18 Generic JSON Serialization

In fact, we rarely need to write instances for the `Decode` class, since the `purescript-foreign-generic` class allows us to *derive* instances using a technique called *datatype-generic programming*. A full explanation of this technique is beyond the scope of this book, but it allows us to write functions once, and reuse them over many different data types, based on the structure of a the types themselves.

To derive a `Decode` instance for our `FormData` type (so that we may deserialize it from its JSON representation), we first use the `derive` keyword to derive an instance of the `Generic` type class, which looks like this:

```
derive instance genericFormData :: Generic FormData _
```

Next, we simply define the `decode` function using the `genericDecode` function, as follows:

```
instance decodeFormData :: Decode FormData where
  decode = genericDecode (defaultOptions { unwrapSingleConstructors = true })
```

In fact, we can also derive an *encoder* in the same way:

```
instance encodeFormData :: Encode FormData where
  encode = genericEncode (defaultOptions { unwrapSingleConstructors = true })
```

It is important that we use the same options in the decoder and encoder, otherwise our encoded JSON documents might not get decoded correctly.

Now, when the Save button is clicked, a value of type `FormData` is passed to the `encode` function, serializing it as a JSON document. The `FormData` type is a newtype for a record, so a value of type `FormData` passed to `encode` will be serialized as a JSON *object*. This is because we used the `unwrapSingleConstructors` option when defining our JSON encoder.

Our `Decode` type class instance is used with `decodeJSON` to parse the JSON document when it is retrieved from local storage, as follows:

```
loadSavedData = do
  item <- getItem "person"

  let
    savedData :: Either (NonEmptyList ForeignError) (Maybe FormData)
    savedData = runExcept do
      jsonOrNull <- traverse readString =<< readNullOrUndefined item
      traverse decodeJSON jsonOrNull
```

The `savedData` action reads the `FormData` structure in two steps: first, it parses the `Foreign` value obtained from `getItem`. The type of `jsonOrNull` is inferred by the compiler to be `Maybe String` (exercise for the reader - how is this type inferred?). The `traverse` function is then used to apply `decodeJSON` to the (possibly missing) element of the result of type `Maybe String`. The type class instance inferred for `decodeJSON` is the one we just wrote, resulting in a value of type `F (Maybe FormData)`.

We need to use the monadic structure of `F`, since the argument to `traverse` uses the result `jsonOrNull` obtained in the first line.

There are three possibilities for the result of `FormData`:

- If the outer constructor is `Left`, then there was an error parsing the JSON string, or it represented a value of the wrong type. In this case, the application displays an error using the `alert` action we wrote earlier.
- If the outer constructor is `Right`, but the inner constructor is `Nothing`, then `getItem` also returned `Nothing` which means that the key did not exist in local storage. In this case, the application continues quietly.
- Finally, a value matching the pattern `Right (Just _)` indicates a successfully parsed JSON document. In this case, the application updates the form fields with the appropriate values.

Try out the code, by running `pulp build -O --to dist/Main.js`, and then opening the browser to `html/index.html`. You should be able to save the form fields' contents to local storage by clicking the Save button, and then see the fields repopulated when the page is refreshed.

*Note*: You may need to serve the HTML and Javascript files from a HTTP server locally in order to avoid certain browser-specific issues.

## Exercises

1. (Easy) Use `decodeJSON` to parse a JSON document representing a two-dimensional JavaScript array of integers, such as `[[1, 2, 3], [4, 5], [6]]`. What if the elements are allowed to be null? What if the arrays themselves are allowed to be null?
2. (Medium) Convince yourself that the implementation of `savedData` should type-check, and write down the inferred types of each subexpression in the computation.
3. (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 `Encode` and `Decode` instances for this type using `purescript-foreign-generic`, and verify that encoded values can correctly be decoded in PSCi.
4. (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 for `Encode` and `Decode` for the `IntOrString` data type which implement this behavior, and verify that encoded values can correctly be decoded in PSCi.

## 10.19 Conclusion

In this chapter, we've learned how to work with foreign JavaScript code from PureScript, and vice versa, and we've seen the issues involved with writing trustworthy code using the FFI:

- We've seen the importance of the *runtime representation* of data, and ensuring that foreign functions have the correct representation.
- We learned how to deal with corner cases like null values and other types of JavaScript data, by using foreign types, or the `Foreign` data type.
- We looked at some common foreign types defined in the Prelude, and how they can be used to interoperate with idiomatic JavaScript code. In particular, the representation of side-effects in the `Eff` monad was introduced, and we saw how to use the `Eff` monad to capture new side effects.
- We saw how to safely deserialize JSON data using the `Decode` type class.

For more examples, the `purescript`, `purescript-contrib` and `purescript-node` GitHub organizations provide plenty of examples of libraries which 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.

# 11. Monadic Adventures

## 11.1 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 which 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.

## 11.2 Project Setup

This module's project introduces the following new Bower dependencies:

- `purescript-maps`, which provides a data type for immutable maps
- `purescript-sets`, which provides a data type for immutable sets
- `purescript-transformers`, which provides implementations of standard monad transformers
- `purescript-node-readline`, which provides FFI bindings to the `readline`[1] interface provided by NodeJS
- `purescript-yargs`, which provides an applicative interface to the `yargs`[2] command line argument processing library

It is also necessary to install the `yargs` module using NPM:

```
npm install
```

## 11.3 How To Play The Game

To run the project, use `pulp run`

By default you will see a usage message:

[1] http://nodejs.org/api/readline.html

[2] https://www.npmjs.org/package/yargs

```
node ./dist/Main.js -p <player name>

Options:
  -p, --player  Player name  [required]
  -d, --debug   Use debug mode

Missing required arguments: p
The player name is required
```

Provide the player name using the `-p` option:

```
pulp run -- -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 be used to 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 which 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:

```
$ pulp run -- -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 `purescript-transformers` package to build a library which will enable rapid development of this type of game.

## 11.4 The State Monad

We will start by looking at some of the monads provided by the `purescript-transformers` package.

The first example is the `State` monad, which provides a way to model *mutable state* in pure code. We have already seen two approaches to mutable state provided by the `Eff` monad, namely the `REF` and `ST` effects. `State` provides a third alternative, but it is not implemented using the `Eff` monad.

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
put    :: forall s. s        -> State s Unit
modify :: forall s. (s -> s) -> State s Unit
```

This looks very similar to the API provided by the `REF` and `ST` effects. However, notice that we do not pass a mutable reference cell such as a `Ref` or `STRef` to the actions. The difference between `State` and the solutions provided by the `Eff` monad is that the `State` monad only supports a single piece of state which is implicit - the state is implemented as a function argument hidden by the `State` monad's data constructor, so there is no explicit reference to pass around.

Let's see an example. One use of the `State` monad might be to add the values in an array of numbers to the current state. We could do that by choosing `Number` 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 Number -> State Number 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 of these functions 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 by 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 which 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` module to turn the input string into an array of characters.

## 11.5 The Reader Monad

Another monad provided by the `purescript-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` which 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.
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"
      ]
  ]
```

## 11.6 The Writer Monad

The `Writer` monad provides the ability to accumulate 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 actually be used to 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` which 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 by 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 `purescript-monoid` package.
2. (Medium) The *Collatz* function is defined on natural numbers `n` as `n / 2` when `n` is even, and `3 * n + 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 which 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.

## 11.7 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 `Eff` monad provides a partial solution to this problem, since native effects can be interleaved using the approach of *extensible effects*. Monad transformers provide another solution, and each approach has its own benefits and limitations.

A monad transformer is a type constructor which is parameterized not only by a type, but by 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 which can fail with an error, and which can use mutable state.

We can use the actions of the outer `StateT String` monad (`get`, `put`, and `modify`) directly, but in order 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`, and `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 `purescript-parsing` library. This is the power of monad transformers - we can create custom-built monads for a variety of problems, choosing the side-effects that we need, and keeping the expressiveness of do notation and applicative combinators.

## 11.8 The ExceptT Monad Transformer

The `purescript-transformers` package also defines the `ExceptT e` monad transformer, which is the 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 monadErrorExceptT :: 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 which support throwing and catching of errors of some type `e`, and an instance is provided for the `ExceptT e` monad transformer. The `throwError` action can be used to indicate failure, just 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 `purescript-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.Trans
import Control.Monad.Writer
import Control.Monad.Writer.Class
import Control.Monad.Error.Class
import Control.Monad.Except.Trans

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 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 which were written before the error was thrown actually get appended to the log.

## 11.9 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 that 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 `Eff` 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. In fact, 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 " <> show 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 `runIdentity`.

```
> runParser p s = runIdentity $ 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 the way in which 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 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.

## 11.10 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 `purescript-transformers` package. Each of these monad transformers 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. `purescript-transformers` defines a type class for each of the major transformers, allowing us to abstract over monads which 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 really 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.

## 11.11 Alternatives

The `purescript-control` package defines a number of abstractions for working with computations which 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.List` module provides two useful functions for working with type constructors in the `Alternative` type class:

```
many :: forall f a. Alternative f => Lazy (f (List a)) => f a -> f (List a)
some :: forall f a. Alternative f => Lazy (f (List a)) => f a -> f (List a)
```

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 Split
> import Control.Alternative

> 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.

Other examples of `Alternative` type constructors are `Maybe` and `Array`.

## 11.12 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) <= MonadZero m

class MonadZero 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. MonadZero 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*.

## 11.13 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 that it should.
2. (Medium) Use your `string` parser with the `many` combinator to write a parser which 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 which 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?

## 11.14 The RWS Monad

One particular combination of monad transformers is so common that it is provided as a single monad transformer in the `purescript-transformers` package. The `Reader`, `Writer` and `State` monads are combined into the *reader-writer-state* monad, or more simply the `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 in terms of 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 which 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 qualified Data.Map as M
import qualified 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, 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
```

## 11.15 Implementing Game Logic

Our game is going to 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 Eff 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 <- 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, then 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.

## 11.16 Running the Computation

Since our game logic runs in the RWS monad, it is necessary to run the computation in order to respond to the user's commands.

The front-end of our game is built using two packages: purescript-yargs, which provides an applicative interface to the yargs command line parsing library, and purescript-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
  :: forall eff
   . GameEnvironment
  -> Eff ( exception :: EXCEPTION
         , readline :: RL.READLINE
         , console :: CONSOLE
         | eff
         ) Unit
```

The CONSOLE effect indicates that this function interacts with the user via the console (using the purescript-node-readline and purescript-console packages). runGame takes the game configuration as a function argument.

The purescript-node-readline package provides the LineHandler type, which represents actions in the Eff monad which handle user input from the terminal. Here is the corresponding API:

```
type LineHandler eff a = String -> Eff eff a

setLineHandler
  :: forall eff a
   . Interface
  -> LineHandler (readline :: READLINE | eff) a
  -> Eff (readline :: READLINE | eff) 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:

```
runGame env = do
  interface <- createConsoleInterface 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:

```
  setPrompt "> " 2 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
  -> Eff ( exception :: EXCEPTION
         , console :: CONSOLE
         , readline :: RL.READLINE
         | eff
         ) Unit
lineHandler currentState input = do
  case runRWS (game (split " " input)) env currentState of
    RWSResult state _ written -> do
      for_ written log
      setLineHandler interface $ lineHandler state
  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:

```
  setLineHandler interface $ lineHandler initialGameState
  prompt interface
```

## Exercises

1. (Medium) Implement a new command `cheat`, which moves all game items from the game grid into the user's inventory.
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.

## 11.17 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 `purescript-yargs` package.

`purescript-yargs` 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 `purescript-yargs` package, the functor we are interested in is the `Y` functor, which adds the side-effect of reading from command line options. It provides the following handler:

```
runY :: forall a eff.
          YargsSetup ->
          Y (Eff (exception :: EXCEPTION, console :: CONSOLE | eff) a) ->
             Eff (exception :: EXCEPTION, console :: CONSOLE | eff) a
```

This is best illustrated by example. The application's `main` function is defined using `runY` as follows:

```
main = runY (usage "$0 -p <player name>") $ map runGame env
```

The first argument is used to configure the `yargs` library. In our case, we simply provide a usage message, but the `Node.Yargs.Setup` module provides several other options.

The second argument uses the `map` function to lift the `runGame` function over the `Y` type constructor. The argument `env` is constructed in a `where` declaration using the applicative operators `<$>` and `<*>`:

```
  where
  env :: Y GameEnvironment
  env = gameEnvironment
          <$> yarg "p" ["player"]
                   (Just "Player name")
                   (Right "The player name is required")
                   false
          <*> flag "d" ["debug"]
                   (Just "Use debug mode")
```

Here, the `gameEnvironment` function, which has the type `PlayerName -> Boolean -> GameEnvironment`, is lifted over `Y`. The two arguments specify how to read the player name and debug flag from the command line options. The first argument describes the player name option, which is specified by the `-p` or `--player` options, and the second describes the debug mode flag, which is turned on using the `-d` or `--debug` options.

This demonstrates two basic functions defined in the `Node.Yargs.Applicative` module: `yarg`, which defines a command line option which takes an optional argument (of type `String`, `Number` or `Boolean`), and `flag` which defines a command line flag of type `Boolean`.

Notice how we were able to use 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, simply 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 `yargs` configuration which enables cheat mode. The `cheat` command from the previous exercise should be disallowed if cheat mode is not enabled.

## 11.18 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 `Eff` 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 `Eff` 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 effects are tracked by the type system. In addition, type classes provide a powerful way to abstract over the actions provided by a monad, enabling code reuse. We were able to use standard abstractions like `Alternative` and `MonadPlus` to build useful monads by combining standard monad transformers.

Monad transformers are an excellent demonstration of the sort of expressive code that can be written by relying on advanced type system features such as higher-kinded polymorphism and multi-parameter type classes.

In the next chapter, we will see how monad transformers can be used to give an elegant solution to a common complaint when working with asynchronous JavaScript code - the problem of *callback hell*.

# 12. Callback Hell

## 12.1 Chapter Goals

In this chapter, we will see how the tools we have seen so far - namely monad transformers and applicative functors - can be put to use to solve real-world problems. In particular, we will see how we can solve the problem of *callback hell*.

## 12.2 Project Setup

The source code for this chapter can be compiled and run using `pulp run`. It is also necessary to install the `request` module using NPM:

```
npm install
```

## 12.3 The Problem

Asynchronous code in JavaScript typically uses *callbacks* to structure program flow. For example, to read text from a file, the preferred approach is to use the `readFile` function and to pass a callback - a function that will be called when the text is available:

```
function readText(onSuccess, onFailure) {
  var fs = require('fs');
  fs.readFile('file1.txt', { encoding: 'utf-8' }, function (error, data) {
    if (error) {
      onFailure(error.code);
    } else {
      onSuccess(data);
    }
  });
}
```

However, if multiple asynchronous operations are involved, this can quickly lead to nested callbacks, which can result in code which is difficult to read:

```
function copyFile(onSuccess, onFailure) {
  var fs = require('fs');
  fs.readFile('file1.txt', { encoding: 'utf-8' }, function (error, data1) {
    if (error) {
      onFailure(error.code);
    } else {
      fs.writeFile('file2.txt', data, { encoding: 'utf-8' }, function (error) {
        if (error) {
          onFailure(error.code);
        } else {
          onSuccess();
        }
      });
    }
  });
}
```

One solution to this problem is to break out individual asynchronous calls into their own functions:

```
function writeCopy(data, onSuccess, onFailure) {
  var fs = require('fs');
  fs.writeFile('file2.txt', data, { encoding: 'utf-8' }, function (error) {
    if (error) {
      onFailure(error.code);
    } else {
      onSuccess();
    }
  });
}

function copyFile(onSuccess, onFailure) {
  var fs = require('fs');
  fs.readFile('file1.txt', { encoding: 'utf-8' }, function (error, data) {
    if (error) {
      onFailure(error.code);
    } else {
      writeCopy(data, onSuccess, onFailure);
    }
  });
}
```

This solution works but has some issues:

- It is necessary to pass intermediate results to asynchronous functions as function arguments, in the same way that we passed `data` to `writeCopy` above. This is fine for small functions, but if there are

many callbacks involved, the data dependencies can become complex, resulting in many additional function arguments.
- There is a common pattern - the callbacks `onSuccess` and `onFailure` are usually specified as arguments to every asynchronous function - but this pattern has to be documented in module documentation which accompanies the source code. It is better to capture this pattern in the type system, and to use the type system to enforce its use.

Next, we will see how to use the techniques we have learned so far to solve these issues.

## 12.4 The Continuation Monad

Let's translate the `copyFile` example above into PureScript by using the FFI. In doing so, the structure of the computation will become apparent, and we will be led naturally to a monad transformer which is defined in the `purescript-transformers` package - the continuation monad transformer `ContT`.

*Note*: in practice, it is not necessary to write these functions by hand every time. Asynchronous file IO functions can be found in the `purescript-node-fs` and `purescript-node-fs-aff` libraries.

First, we need to gives types to `readFile` and `writeFile` using the FFI. Let's start by defining some type synonyms, and a new effect for file IO:

```
foreign import data FS :: Effect

type ErrorCode = String
type FilePath = String
```

`readFile` takes a filename and a callback which takes two arguments. If the file was read successfully, the second argument will contain the file contents, and if not, the first argument will be used to indicate the error.

In our case, we will wrap `readFile` with a function which takes two callbacks: an error callback (`onFailure`) and a result callback (`onSuccess`), much like we did with `copyFile` and `writeCopy` above. Using the multiple-argument function support from `Data.Function` for simplicity, our wrapped function `readFileImpl` might look like this:

```
foreign import readFileImpl
  :: forall eff
   . Fn3 FilePath
       (String -> Eff (fs :: FS | eff) Unit)
       (ErrorCode -> Eff (fs :: FS | eff) Unit)
       (Eff (fs :: FS | eff) Unit)
```

In the foreign Javascript module, `readFileImpl` would be defined as:

```
exports.readFileImpl = function(path, onSuccess, onFailure) {
  return function() {
    require('fs').readFile(path, {
      encoding: 'utf-8'
    }, function(error, data) {
      if (error) {
        onFailure(error.code)();
      } else {
        onSuccess(data)();
      }
    });
  };
};
```

This type signature indicates that `readFileImpl` takes three arguments: a file path, a success callback and an error callback, and returns an effectful computation which returns an empty (`Unit`) result. Notice that the callbacks themselves are given types which use the `Eff` monad to track their effects.

You should try to understand why this implementation has the correct runtime representation for its type.

`writeFileImpl` is very similar - it is different only in that the file content is passed to the function itself, not to the callback. Its implementation looks like this:

```
foreign import writeFileImpl
  :: forall eff
   . Fn4 FilePath
         String
         (Eff (fs :: FS | eff) Unit)
         (ErrorCode -> Eff (fs :: FS | eff) Unit)
         (Eff (fs :: FS | eff) Unit)
```

```
exports.writeFileImpl = function(path, data, onSuccess, onFailure) {
  return function() {
    require('fs').writeFile(path, data, {
      encoding: 'utf-8'
    }, function(error) {
      if (error) {
        onFailure(error.code)();
      } else {
        onSuccess();
      }
    });
  };
};
```

Given these FFI declarations, we can write the implementations of `readFile` and `writeFile`. These will use the `Data.Function.Uncurried` module to turn the multiple-argument FFI bindings into regular (curried) PureScript functions, and therefore have slightly more readable types.

In addition, instead of requiring two callbacks, one for successes and one for failures, we can require only a single callback which responds to *either* successes or failures. That is, the new callback takes a value in the `Either ErrorCode` monad as its argument:

```
readFile
  :: forall eff
   . FilePath
  -> (Either ErrorCode String -> Eff (fs :: FS | eff) Unit)
  -> Eff (fs :: FS | eff) Unit
readFile path k =
  runFn3 readFileImpl
         path
         (k <<< Right)
         (k <<< Left)

writeFile
  :: forall eff
   . FilePath
  -> String
  -> (Either ErrorCode Unit -> Eff (fs :: FS | eff) Unit)
  -> Eff (fs :: FS | eff) Unit
writeFile path text k =
  runFn4 writeFileImpl
         path
         text
         (k $ Right unit)
         (k <<< Left)
```

Now we can spot an important pattern. Each of these functions takes a callback which returns a value in some monad (in this case `Eff (fs :: FS | eff)`) and returns a value in *the same monad*. This means that when the first callback returns a result, that monad can be used to bind the result to the input of the next asynchronous function. In fact, that's exactly what we did by hand in the `copyFile` example.

This is the basis of the *continuation monad transformer*, which is defined in the `Control.Monad.Cont.Trans` module in `purescript-transformers`.

`ContT` is defined as a newtype as follows:

```
newtype ContT r m a = ContT ((a -> m r) -> m r)
```

A *continuation* is just another name for a callback. A continuation captures the *remainder* of a computation - in our case, what happens after a result has been provided after an asynchronous call.

The argument to the `ContT` data constructor looks remarkably similar to the types of `readFile` and `writeFile`. In fact, if we take the type `a` to be the type `Either ErrorCode String`, `r` to be `Unit` and `m` to be the monad `Eff (fs :: FS | eff)`, we recover the right-hand side of the type of `readFile`.

This motivates the following type synonym, defining an `Async` monad, which we will use to compose asynchronous actions like `readFile` and `writeFile`:

```
type Async eff = ContT Unit (Eff eff)
```

For our purposes, we will always use `ContT` to transform the `Eff` monad, and the type `r` will always be `Unit`, but this is not required.

We can treat `readFile` and `writeFile` as computations in the `Async` monad, by simply applying the `ContT` data constructor:

```
readFileCont
  :: forall eff
   . FilePath
  -> Async (fs :: FS | eff) (Either ErrorCode String)
readFileCont path = ContT $ readFile path

writeFileCont
  :: forall eff
   . FilePath
  -> String
  -> Async (fs :: FS | eff) (Either ErrorCode Unit)
writeFileCont path text = ContT $ writeFile path text
```

With that, we can write our copy-file routine by simply using do notation for the `ContT` monad transformer:

```
copyFileCont
  :: forall eff
   . FilePath
  -> FilePath
  -> Async (fs :: FS | eff) (Either ErrorCode Unit)
copyFileCont src dest = do
  e <- readFileCont src
  case e of
    Left err -> pure $ Left err
    Right content -> writeFileCont dest content
```

Note how the asynchronous nature of `readFileCont` is hidden by the monadic bind expressed using do notation - this looks just like synchronous code, but the `ContT` monad is taking care of wiring our asynchronous functions together.

We can run this computation using the `runContT` handler by providing a continuation. The continuation represents *what to do next*, i.e. what to do when the asynchronous copy-file routine completes. For our simple example, we can just choose the `logShow` function as the continuation, which will print the result of type `Either ErrorCode Unit` to the console:

```
import Prelude

import Control.Monad.Eff.Console (logShow)
import Control.Monad.Cont.Trans (runContT)

main =
  runContT
    (copyFileCont "/tmp/1.txt" "/tmp/2.txt")
    logShow
```

## Exercises

1. (Easy) Use `readFileCont` and `writeFileCont` to write a function which concatenates two text files.
2. (Medium) Use the FFI to give an appropriate type to the `setTimeout` function. Write a wrapper function which uses the `Async` monad:

```
type Milliseconds = Int

foreign import data TIMEOUT :: Effect

setTimeoutCont
  :: forall eff
   . Milliseconds
  -> Async (timeout :: TIMEOUT | eff) Unit
```

## 12.5 Putting ExceptT To Work

This solution works, but it can be improved.

In the implementation of `copyFileCont`, we had to use pattern matching to analyze the result of the `readFileCont` computation (of type `Either ErrorCode String`) to determine what to do next. However, we know that the `Either` monad has a corresponding monad transformer, `ExceptT`, so it is reasonable to expect that we should be able to use `ExceptT` with `ContT` to combine the two effects of asynchronous computation and error handling.

In fact, it is possible, and we can see why if we look at the definition of `ExceptT`:

```
newtype ExceptT e m a = ExceptT (m (Either e a))
```

ExceptT simply changes the result of the underlying monad from a to Either e a. This means that we can rewrite copyFileCont by transforming our current monad stack with the ExceptT ErrorCode transformer. It is as simple as applying the ExceptT data constructor to our existing solution:

```
readFileContEx
  :: forall eff
   . FilePath
  -> ExceptT ErrorCode (Async (fs :: FS | eff)) String
readFileContEx path = ExceptT $ readFileCont path

writeFileContEx
  :: forall eff
   . FilePath
  -> String
  -> ExceptT ErrorCode (Async (fs :: FS | eff)) Unit
writeFileContEx path text = ExceptT $ writeFileCont path text
```

Now, our copy-file routine is much simpler, since the asynchronous error handling is hidden inside the ExceptT monad transformer:

```
copyFileContEx
  :: forall eff
   . FilePath
  -> FilePath
  -> ExceptT ErrorCode (Async (fs :: FS | eff)) Unit
copyFileContEx src dest = do
  content <- readFileContEx src
  writeFileContEx dest content
```

## Exercises

1. (Medium) Modify your solution which concatenated two files, using ExceptT to handle any errors.
2. (Medium) Write a function concatenateMany to concatenate multiple text files, given an array of input file names. *Hint*: use traverse.

## 12.6 A HTTP Client

As another example of using `ContT` to handle asynchronous functions, we'll now look at the `Network.HTTP.Client` module from this chapter's source code. This module uses the `Async` monad to support asynchronous HTTP requests using the `request` module, which is available via NPM.

The `request` module provides a function which takes a URL and a callback, makes a HTTP(S) request and invokes the callback when the response is available, or in the event of an error. Here is an example request:

```
require('request')('http://purescript.org'), function(err, _, body) {
  if (err) {
    console.error(err);
  } else {
    console.log(body);
  }
});
```

We will recreate this simple example in PureScript using the `Async` monad.

In the `Network.HTTP.Client` module, the `request` method is wrapped with a function `getImpl`:

```
foreign import data HTTP :: Effect

type URI = String

foreign import getImpl
  :: forall eff
   . Fn3 URI
         (String -> Eff (http :: HTTP | eff) Unit)
         (String -> Eff (http :: HTTP | eff) Unit)
         (Eff (http :: HTTP | eff) Unit)
```

```
exports.getImpl = function(uri, done, fail) {
  return function() {
    require('request')(uri, function(err, _, body) {
      if (err) {
        fail(err)();
      } else {
        done(body)();
      }
    });
  };
};
```

Again, we can use the `Data.Function.Uncurried` module to turn this into a regular, curried PureScript function. As before, we turn the two callbacks into a single callback, this time accepting a value of type `Either String String`, and apply the `ContT` constructor to construct an action in our `Async` monad:

```
get :: forall eff.
  URI ->
  Async (http :: HTTP | eff) (Either String String)
get req = ContT \k ->
  runFn3 getImpl req (k <<< Right) (k <<< Left)
```

## Exercises

1. (Easy) Use `runContT` to test `get` in PSCi, printing the result to the console.
2. (Medium) Use `ExceptT` to write a function `getEx` which wraps `get`, as we did previously for `readFileCont` and `writeFileCont`.
3. (Difficult) Write a function which saves the response body of a request to a file on disk using `getEx` and `writeFileContEx`.

## 12.7 Parallel Computations

We've seen how to use the `ContT` monad and do notation to compose asynchronous computations in sequence. It would also be useful to be able to compose asynchronous computations *in parallel*.

If we are using `ContT` to transform the `Eff` monad, then we can compute in parallel simply by initiating our two computations one after the other.

The `purescript-parallel` package defines a type class `Parallel` for monads like `Async` 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 `Async`) and an applicative functor `f` which 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 `purescript-parallel` library provides a `Parallel` instance for the `Async` monad. It uses mutable references to combine `Async` 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.

We can use the `parallel` function to create a version of our `readFileCont` action which can be combined in parallel. Here is a simple example which reads two text files in parallel, and concatenates and prints their results:

```
import Prelude
import Control.Apply (lift2)
import Control.Monad.Cont.Trans (runContT)
import Control.Monad.Eff.Console (logShow)
import Control.Monad.Parallel (parallel, sequential)

main = flip runContT logShow do
  sequential $
   lift2 append
     <$> parallel (readFileCont "/tmp/1.txt")
     <*> parallel (readFileCont "/tmp/2.txt")
```

Note that, since `readFileCont` returns a value of type `Either ErrorCode String`, we need to lift the `append` function over the `Either` type constructor using `lift2` to form our combining function.

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.

## Exercises

1. (Easy) Use `parallel` and `sequential` to make two HTTP requests and collect their response bodies in parallel. Your combining function should concatenate the two response bodies, and your continuation should use `print` to print the result to the console.
2. (Medium) The applicative functor which corresponds to `Async` is also an instance of `Alternative`. The `<|>` operator defined by this instance runs two computations in parallel, and returns the result from the computation which completes first.

   Use this `Alternative` instance in conjunction with your `setTimeoutCont` function to define a function

   ```
   timeout :: forall a eff
            . Milliseconds
           -> Async (timeout :: TIMEOUT | eff) a
           -> Async (timeout :: TIMEOUT | eff) (Maybe a)
   ```

   which returns `Nothing` if the specified computation does not provide a result within the given number of milliseconds.
3. (Medium) `purescript-parallel` also provides instances of the `Parallel` class for several monad transformers, including `ExceptT`.

   Rewrite the parallel file IO example to use `ExceptT` for error handling, instead of lifting `append` with `lift2`. Your solution should use the `ExceptT` transformer to transform the `Async` monad.

   Use this approach to modify your `concatenateMany` function to read multiple input files in parallel.
4. (Difficult, Extended) Suppose we are given a collection of JSON documents on disk, such that each document contains an array of references to other files on disk:

   ```
   { references: ['/tmp/1.json', '/tmp/2.json'] }
   ```

   Write a utility which takes a single filename as input, and spiders the JSON files on disk referenced transitively by that file, collecting a list of all referenced files.

   Your utility should use the `purescript-foreign` library to parse the JSON documents, and should fetch files referenced by a single file in parallel.

## 12.8 Conclusion

In this chapter, we have seen a practical demonstration of monad transformers:

- We saw how the common JavaScript idiom of callback-passing can be captured by the `ContT` monad transformer.
- We saw how the problem of callback hell can be solved by using do notation to express sequential asynchronous computations, and applicative functors to express parallelism.
- We used `ExceptT` to express *asynchronous errors*.

# 13. Generative Testing

## 13.1 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[1] library in Haskell.

The `purescript-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 `purescript-quickcheck` to test a simple library, using Pulp to integrate our test suite into our development process.

## 13.2 Project Setup

This chapter's project adds `purescript-quickcheck` as a Bower dependency.

In a Pulp 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 `pulp test` command.

## 13.3 Writing Properties

The `Merge` module implements a simple function `merge`, which we will use to demonstrate the features of the `purescript-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, 6]

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

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 in two properties:

[1] http://www.haskell.org/haskellwiki/Introduction_to_QuickCheck1

- (Sortedness) If `xs` and `ys` are sorted, then `merge xs ys` is also sorted.
- (Subarray) `xs` and `ys` are both subarrays of `merge xs ys`, and their elements appear in the same order.

`purescript-quickcheck` allows us to test these properties directly, by generating random test cases. We simply state the properties that we want our code to have, as functions:

```
main = do
  quickCheck \xs ys ->
    isSorted $ merge (sort xs) (sort ys)
  quickCheck \xs ys ->
    xs `isSubarrayOf` merge xs ys
```

Here, `isSorted` and `isSubarrayOf` are implemented as helper functions with the following types:

```
isSorted :: forall a. Ord a => Array a -> Boolean
isSubarrayOf :: forall a. Eq a => Array a -> Array a -> Boolean
```

When we run this code, `purescript-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:

```
$ pulp test

* Build successful. Running tests...

100/100 test(s) passed.
100/100 test(s) passed.

* Tests OK.
```

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.

## 13.4 Improving Error Messages

To provide error messages along with our failed test cases, `purescript-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)
  in
    xs `isSubarrayOf` result <?> show xs <> " not a subarray of " <> show result
```

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 6 failed:
[79168] not a subarray of [-752832,686016]
```

Notice how the input `xs` and `ys` were generated as a arrays of randomly-selected integers.

## Exercises

1. (Easy) Write a property which asserts that merging an array with the empty array does not modify the original array.
2. (Easy) Add an appropriate error message to the remaining property for `merge`.

## 13.5 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 tests 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 a helper function to force the compiler to infer a particular type. 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 tests so that the compiler infers the type `Array Int` for our two array arguments:

```
quickCheck \xs ys ->
  isSorted $ ints $ mergePoly (sort xs) (sort ys)
quickCheck \xs ys ->
  ints xs `isSubarrayOf` mergePoly xs ys
```

Here, `xs` and `ys` both have type `Array Int`, since the `ints` function has been used to disambiguate the unknown type.

### Exercises

1. (Easy) Write a function `bools` which forces the types of `xs` and `ys` to be `Array Boolean`, and add additional properties which test `mergePoly` at that type.
2. (Medium) Choose a pure function from the core libraries (for example, from the `purescript-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`.

## 13.6 Generating Arbitrary Data

Now we will see how the `purescript-quickcheck` library is able to 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 `purescript-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 arbitraryByte :: 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 sortedness test for `merge`:

```
quickCheck \xs ys ->
  isSorted $ numbers $ mergePoly (sort xs) (sort 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 which generates sorted data:

```
newtype Sorted a = Sorted (Array a)

sorted :: forall a. Sorted a -> Array a
sorted (Sorted xs) = xs

instance arbSorted :: (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 ->
  isSorted $ ints $ mergePoly (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 is used to insert a new element into a sorted tree, and the `member` function can be used to 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 be used to convert sorted trees to and from arrays. We can use `fromArray` to write an `Arbitrary` instance for trees:

```
instance arbTree :: (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 which asserts that a value inserted into a tree is still a member of that tree after arbitrarily many more insertions.

## 13.7 Testing Higher-Order Functions

The `Merge` module defines another generalization of the `merge` function - the `mergeWith` function takes an additional function as an argument which is used 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 which are 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 which 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`, and 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 which gives us `Arbitrary` functions if the function domain is `Coarbitrary` and the function codomain is `Arbitrary`:

```
instance arbFunction :: (Coarbitrary a, Arbitrary b) => Arbitrary (a -> b)
```

In practice, this means that we can write properties which 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.

In the case of the sortedness property, 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 with respect to `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 ->
  isSorted $
    map f $
      mergeWith (intToBool f)
                (sortBy (compare `on` f) xs)
                (sortBy (compare `on` f) ys)
```

Here, we use a function `intToBool` to disambiguate the type of the function `f`:

```
intToBool :: (Int -> Boolean) -> Int -> Boolean
intToBool = id
```

In the case of the subarray property, we simply have to change the name of the function to `mergeWith` - we still expect our input arrays to be subarrays of the result:

```
quickCheck \xs ys f ->
  xs `isSubarrayOf` mergeWith (numberToBool f) xs ys
```

In addition to being `Arbitrary`, functions are also `Coarbitrary`:

```
instance coarbFunction :: (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.

## 13.8 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 coarbTree :: Coarbitrary a => Coarbitrary (Tree a) where
```

We have to write a function which perturbs a random generator given a value of type `Tree a`. If the input value is a `Leaf`, then we will just 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, using function composition to create our perturbing function:

```
  coarbitrary (Branch l a r) =
    coarbitrary l <<<
    coarbitrary a <<<
    coarbitrary r
```

Now we are free to 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 are able to 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
```

## 13.9 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 which 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 generating 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 the `all` function to simplify the result of the `quickCheckPure` function - your function should return `true` if every test passes, and `false` otherwise. Try using the `First` monoid, defined in `purescript-monoids` with the `foldMap` function to preserve the first error in case of failure.

## 13.10 Conclusion

In this chapter, we met the `purescript-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 `pulp 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.

# 14. Domain-Specific Languages

## 14.1 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 which is well-suited to development in a particular problem domain. Its syntax and functions are chosen to maximize readability of code used to express ideas in that domain. We have already seen a number of 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 library of combinators which we wrote for the `Async` and `Parallel` functors in chapter 12 could be considered an example of a domain-specific language for the domain of *asynchronous programming*.
- The `purescript-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 of standard techniques in the implementation of 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. Our aim will be to develop a type-safe language for describing correct HTML documents, and we will work by improving a naive implementation in small steps.

## 14.2 Project Setup

The project accompanying this chapter adds one new Bower dependency - the `purescript-free` library, which defines the *free monad*, one of the tools which we will be using.

We will test this chapter's project in PSCi.

## 14.3 A 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:

```
$ pulp repl

> import Prelude
> import Data.DOM.Simple
> import Data.Maybe
> import Control.Monad.Eff.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.

## 14.4 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 which is 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 would be able to 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 which do not exist or whose names were entered incorrectly. To solve this problem, we can create a newtype which 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:

```
$ pulp repl

> import Prelude
> import Data.DOM.Smart
> import Control.Monad.Eff.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 which is 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.

## 14.5 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 stringIsValue :: IsValue String where
  toValue = id

instance intIsValue :: 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 Control.Monad.Eff.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 which represents 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.

## 14.6 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 `purescript-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`. In fact, we can simply modify our `Content` type as follows:

```
data ContentF a
  = TextContent String a
  | ElementContent Element a

instance functorContentF :: 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 simply 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.

## 14.7 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 using monads `m` which satisfy the stronger `MonadRec` constraint. In practice, this means that 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 a 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 simply 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 Control.Monad.Eff.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.

## 14.8 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 which returns a non-trivial result.

Suppose we want to generate HTML documents which contain hyperlinks to different sections of the document using *anchors*. We can accomplish this already, by generating anchor names by hand and including them at least twice in the document: once at the definition of the anchor itself, 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.

In the interest of protecting the developer from their own mistakes, we can introduce a new type which 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 are able to use names in attribute values:

```
instance nameIsValue :: 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 hrefIsValue :: 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 functorContentF :: 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 that 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 does not have the ability to 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`. We do, however, 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 Control.Monad.Eff.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 by 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.

## 14.9 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 which were *correct-by-construction*.
- We used an *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.

The implementation of domain-specific languages in functional programming languages is an area of active research, but hopefully this provides a useful introduction some simple techniques, and illustrates the power of working in a language with expressive types.