PureScript By Example

Functional Programming for the Web

By 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 2014-08-05

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:

https://twitter.com/search?q=#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");
      }
    });
  }
});
```

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.

[1] http://underscorejs.org

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.

1.2 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
```

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, I'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 built from source on any system running an up-to-date installation of the Haskell Platform, and we will walk through this process in the next chapter.

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, Grunt, and Bower, 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 recognise 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[2]. 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

main = Debug.Trace.trace "Hello, World!"
```

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

```
$ psc src/Main.purs
```

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:

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

```
> 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.

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[3] contains blog posts written by the PureScript developers, as well as links to videos and other resources for beginners.
- The PureScript compiler documentation[4] contains simple code examples for the major features of the language.
- Try PureScript![5] is a website which allows users to compile PureScript code in the web browser, and contains several simple examples of code.

If you prefer to learn by reading examples, the `purescript` and `purescript-contrib` GitHub organisations 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 (Java, Scala, C#, F#).

Not long into my professional career, I began to appreciate functional programming and its connections with mathematics, and fell in love with 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,

[3] http://purescript.org
[4] http://docs.purescript.org
[5] http://try.purescript.org

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[6], and can be reached on Twitter[7].

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.

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[8].

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

[6] http://functorial.com

[7] http://twitter.com/paf31

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

2 Getting Started

2.1 Chapter Goals

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

The first code we will write is an example of a library which will use dependencies from NPM and Bower, and which will be built using the Grunt build automation tool. It will provide a single function, to 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:

- `psc`[1] - The PureScript compiler itself.
- `npm`[2] - The Node Package Manager, which will allow us to install the rest of our development tools.
- `bower`[3] - A package manager which is used to version various PureScript packages which we will need.
- `grunt`[4] - An automation tool which we will use to build our PureScript code.

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 build the compiler from source. The PureScript compiler can be downloaded as a binary distribution for 64-bit Ubuntu from the PureScript website[5], but binary distributions are currently only made for major releases. If you would like to stay up-to-date with the latest bug fixes and feature additions, and ensure that the compiler can build the latest packages, you should follow these instructions to build the latest minor release.

[1] http://purescript.org
[2] http://npmjs.org
[3] http://bower.io/
[4] http://gruntjs.com/
[5] http://purescript.org

The main requirement is a working installation of the Haskell Platform[6]. Depending on your operating system, you may also need to install the `ncurses` development package using your package manager (available in Ubuntu as the `libncurses5-dev` package, for example).

Begin by ensuring that you have a recent version of the Cabal executable installed:

```
$ cabal install Cabal cabal-install
```

Also make sure the Cabal package list is up-to-date:

```
$ cabal update
```

The PureScript compiler can either be installed globally, or in a Cabal sandbox in a local directory. This section will cover how to install PureScript globally so that its executables are available on your path.

Install PureScript from Hackage using the `cabal install` command:

```
$ cabal install purescript
```

The compiler and associated executables will now be available on your path. Try running the PureScript compiler on the command line to verify this:

```
$ psc
```

2.4 Installing Tools

If you do not have a working installation of NodeJS[7], 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.

Once you have a working copy of `npm` installed, you will need to install Grunt and Bower. It is usually a good idea to install these globally, so that their command line tools will be available to you regardless of which project you are working in.

```
$ npm install -g grunt-cli bower
```

At this point, you will have all the tools needed to create your first PureScript project.

[6] http://haskell.org/platform
[7] http://nodejs.org/

2.5 Hello, PureScript!

Let's start out simple. We'll use the PureScript compiler `psc` directly to compile a basic Hello World! program. As the chapter progresses, we'll automate more and more of the development process, until we can build our app from scratch including all dependencies with three standard commands.

First of all, create a directory `src` for your source files, and paste the following into a file named `src/Chapter2.purs`:

```
module Chapter2 where

import Debug.Trace

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

This small sample illustrates a few key ideas:

- 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 `Debug.Trace` module, which provides the `trace` 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. Invoke the following command:

```
$ psc src/Chapter2.purs
```

If everything worked, then you will see a relatively large amount of Javascript emitted onto the console. Instead, let's redirect the output to a file with the `--output` command line option:

```
$ psc src/Chapter2.purs --output dist/Main.js
```

You should now be able to run your code using NodeJS:

```
$ node dist/Main.js
```

If that worked, NodeJS should execute your code, and correctly print nothing to the console. The reason is that we have not told the PureScript compiler the name of our main module!

```
$ psc src/Chapter2.purs --output dist/Main.js --main Chapter2
```

This time, if you run run your code, you should see the words “Hello, World!” printed to the console.

2.6 Removing Unused Code

If you open the `dist/Main.js` file in a text editor, you will see quite a large amount of JavaScript. The reason for this is that the compiler ships with a set of standard functions in a set of modules called the Prelude. The Prelude includes the `Debug.Trace` module that we are using to print to the console.

In fact, almost none of this generated code is being used, and we can remove the unused code with another compiler option:

```
$ psc src/Chapter2.purs --output dist/Main.js --main Chapter2 --module Chapter2
```

I’ve added the `--module Chapter2` option, which tells `psc` only to include JavaScript which is required by the code defined in the `Chapter2` module. This time, if you open the generated code in a text editor, you should see the following:

```
var PS = PS || {};
PS.Debug_Trace = (function () {
    "use strict";
    function trace(s) {
      return function() {
        console.log(s);
        return {};
      };
    };
    return {
        trace: trace
    };
})();

var PS = PS || {};
PS.Chapter2 = (function () {
    "use strict";
    var Debug_Trace = PS.Debug_Trace;
    var main = Debug_Trace.trace("Hello, World!");
    return {
        main: main
```

```
    };
})();

PS.Chapter2.main();
```

If you run this code using NodeJS, you should see the same text printed onto the console.

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

- Every module gets turned into a 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.

2.7 Automating the Build with Grunt

Now let's set up Grunt to build our code for us, instead of having to type out the PureScript compiler options by hand every time.

Create a file in the project directory called `Gruntfile.js` and paste the following code:

```
module.exports = function(grunt) {

  "use strict";

  grunt.initConfig({

    psc: {
      options: {
        main: "Chapter2",
        modules: ["Chapter2"]
      },
```

```
      all: {
        src: ["src/**/*.purs"],
        dest: "dist/Main.js"
      }
    }
  });

  grunt.loadNpmTasks("grunt-purescript");

  grunt.registerTask("default", ["psc:all"]);
};
```

This file defines a Node module, which uses the `grunt` module as a library to define a build configuration. It uses the `grunt-purescript` plugin, which invokes the PureScript compiler and exposes its command line options as JSON properties.

The `grunt-purescript` plugin also provides other useful capabilities, such as the ability to automatically generate Markdown documentation from your code, or generate configuration files for your libraries for the `psci` interactive compiler. The interested reader is referred to the `grunt-purescript` project homepage[8].

Install the `grunt` library and the `grunt-purescript` plugin into your local modules directory as follows:

```
$ npm install grunt grunt-purescript@0.5.1
```

With the `Gruntfile.js` file saved, you can now compile your code as follows:

```
$ grunt
>> Created file dist/Main.js.

Done, without errors.
```

2.8 Creating an NPM Package

Now that you've set up Grunt, you don't have to type out compiler commands every time you want to recompile, but more importantly, the end-users of your code don't need to either. However, we've now added an extra step: we need to install a custom set of NPM packages before we can build.

Let's define an NPM package of our own, which specifies our dependencies.

In the project directory, run the `npm` executable, specifying the `init` subcommand, to initialize a new project:

[8] http://github.com/purescript-contrib/grunt-purescript

```
$ npm init
```

You will be asked a variety of questions, at the end of which, a file named `package.json` will be added to the project directory. This file specifies our project properties, and we can add our dependencies as an additional property. Open the file in a text editor, and add the following property to the main JSON object:

```
dependencies: {
  "grunt-purescript": "0.5.1"
}
```

This specifies an exact version of the `grunt-purescript` plugin that we'd like to install.

Now, instead of having to install dependencies by hand, your end-users can simply use `npm` to install everything that is required:

```
$ npm install
```

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. Just like we did with our `npm` dependencies, we could download this package directly on the command line, by typing:

```
$ bower install purescript-math#0.1.0
```

This will install version 0.1.0 of the `purescript-math` library, along with its dependencies.

However, we can set up a `bower.json` file which contains our Bower dependencies, just like we used `npm init` to create `package.json` and control our NPM dependencies.

On the command line, run:

```
$ bower init
```

Just like in the case of NPM, you will be asked a collection of questions, at the end of which, a `bower.json` file will be placed in the project directory. During this process, you will be asked whether you would like to include existing dependencies in the project file. If you select Yes, you should see a section like this in `bower.json`:

```
dependencies: {
  'purescript-math': '0.1.0'
}
```

Now, your users will not have to specify dependencies by hand, but instead can pull dependendencies by simply invoking:

```
$ bower update
```

Let's update our Grunt script to include dependencies pulled from Bower. Edit `Gruntfile.js` to change the source files line as follows:

```
src: ["src/**/*.purs", "bower_components/**/src/**/*.purs"]
```

This line includes source files from the `bower_components` directory. If you have a custom Bower configuration, you may have to change this line accordingly.

NPM or Bower?

You may be asking yourself: why do we need to use two package managers? Can't the PureScript libraries be included in the NPM registry?

The PureScript community has standardised on using Bower for PureScript dependencies for a number of reasons:

- PureScript library packages rarely contain JavaScript source code, so are not suitable for deployment into the NPM registry without being compiled first.
- The Bower registry simply maintains a mapping from package names and versions to existing Git repositories, instead of hosting code directly. This allows the community to use existing tools such as GitHub to manage code and releases.
- Bower does not require packages to conform to any particular layout, such as the CommonJS module standard.

Of course, you are free to use any package manager of your choice - the PureScript compiler and tools are not dependent on Bower (or NPM or Grunt, for that matter) in any way.

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/Chapter2.purs` file:

```
import Math
```

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 = print (diagonal 3 4)
```

Now compile the module again, using Grunt:

```
$ grunt
```

If you run the generated code again, you should see that your code has been invoked successfully:

```
$ node dist/Main.js

5
```

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.

The `grunt-purescript` plugin can be configured to generate a `psci` configuration for your source files. This saves you the trouble of having to load your modules into `psci` manually.

To set this up, add a new build target to your `Gruntfile.js` file, below the `psc` or `pscMake` target:

```
dotPsci: ["src/**/*.purs", "bower_components/**/src/**/*.purs"]
```

Also, add this target to the default task:

```
grunt.registerTask("default", ["psc:all", "dotPsci"]);
```

Now, if you run `grunt`, a `.psci` file will be generated in the project directory. This file specifies the commands which should be used to configure `psci` when the interactive mode is loaded.

Load `psci` now:

```
$ psci
>
```

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

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

    :?              Show this help menu
    :i <module>     Import <module> for use in PSCI
    :m <file>       Load <file> for importing
    :q              Quit PSCi
    :r              Reset
    :t <expr>       Show the type of <expr>
```

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.

Try evaluating a few expressions now. To evaluate an expression in `psci`, type one or more lines, terminated with Ctrl+D:

```
> 1 + 2
3

> "Hello, " ++ "World!"
"Hello, World!"
```

Let's try out our new `diagonal` function in `psci`:

```
> Chapter2.diagonal 8 12

13
```

You can also use `psci` to define functions:

```
> let square x = x * x

> square 10
100
```

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 `:t` command:

```
> :t true
Prim.Boolean

> :t [1, 2, 3]
[Prim.Number]
```

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

2.12 Optional: Building CommonJS Modules

The PureScript compiler `psc` generates JavaScript code in a single output file, which is suitable for use in a web browser. There is another option for compilation, called `psc-make`, which can generate a separate CommonJS module for every PureScript module which is compiled. This may be preferable if you are targetting a runtime like NodeJS which supports the CommonJS module standard.

To invoke `psc-make` on the command line, specify the input files, and a directory in which CommonJS modules should be created with the `--output` option:

```
$ psc-make src/Chapter2.purs --output dist/
```

This will create a subdirectory inside the `dist/` directory for every module provided as an input file. If you are using Bower dependencies, don't forget to include source files in the `bower_components/` directory as well!

The `grunt-purescript` plugin also supports compilation using `psc-make`. To use `psc-make` from Grunt: make the following changes in the `Gruntfile.js` file:

- Change the build target from `psc` to `pscMake`.
- Change the destination from a single file `dist/Main.js` to a directory: `dest: "dist/"`
- Change the default task to reference the `pscMake` build target.

Now, the `grunt` command line tool should create a subdirectory under `dist/` for the `Chapter2` module and every one of its dependencies.

2.13 Using Grunt Project Templates

NPM, Grunt, and Bower can be customized in many ways, to support a variety of interesting build processes. However, for simple projects, the steps can be automated by using a Grunt project template.

The `grunt-init` tool provides a way to bootstrap a simple project from a template. The `grunt-init-purescript` project provides a simple template for a PureScript project, including a simple test suite.

To set up a project using `grunt-init`, first install the command line tool using NPM:

```
$ npm install -g grunt-init
```

Now, clone the PureScript template into your home directory. For example, on Linux, or Mac:

```
$ mkdir ~/.grunt-init
$ git clone https://github.com/purescript-contrib/grunt-init-purescript.git \
    ~/.grunt-init/purescript
```

Now, you can create a simple project in a new directory:

```
$ mkdir new-project
$ cd new-project/
$ grunt-init purescript
```

You will be asked a number of simple questions, after which the project will be initialized in the current directory, and ready to build, using the commands we have already seen:

```
$ npm install
$ bower update
$ grunt
```

The final command will build the source files, and run the test suite.

You can use this project template as the basis of a more complicated project.

Exercises

1. (Easy) Use the `Math.pi` constant to write a function `circleArea` which computes the area of a circle with a given radius. Test your function using `psci`.
2. (Medium) Add a Grunt task to the `Gruntfile.js` file to execute the compiled code using NodeJS, so that instead of typing `node dist/Main.js`, the user can simply type `grunt run`. *Hint*: Consider using the `grunt-execute` Grunt plugin.

2.14 Conclusion

In this chapter, we set up a development environment from scratch, using standard tools from the JavaScript ecosystem: NPM, Bower and Grunt.

We also wrote our first PureScript function, and a JavaScript program which could be compiled and executed using 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.