PRAGMATIC FUNCTIONAL JAVASCRIPT

by

Marcelo Camargo

Table of Contents

Introduction

This book is separated in 15 chapters with specific subsections:

1. **Introduction**: gives you a brief introduction about pragmatic software development, functional programming history and reason and how it is applicable to modern software development.
2. **Roots of the evil**: presents language features that are more harmful than beneficial and later focuses on alternatives to solve the problems introduced by them.
3. **Meet ESLint**: shows how to encourage writing functional code and restricting harmful language features by using a specific tool for that job.
4. **Modules**: focuses on how to organize modular code and functions according to their domain and structuring the file system of your program. There is a special focus on modular code, code that is not specific for the program, but that can be easily reused and help you writing less.
5. **The power of composition**: touches on what really functional programming is about and shows you how to create more complexes things by composing very simple ones. It is definitely like playing LEGO!
6. **Types**: provides an overview about how a type system behaves and tools for static type checking on JavaScript in order to reduce the number of bugs. It also provides a different overview about static type system, different from what is presented in most courses and tutorials.
7. **Transforming values**: talks about common data structures transformations and functions that help us modeling transformations and working with immutable data structures. We'll be using a library called Ramda to present it in practice.
8. **Monads, monoids and functors**: may look a bit theoretical, but will present, together with the concepts of the terms, the practical usage of them and how important they are on the design of a functional program and how they help us to handle errors and deal with the dangerous world that exist outside.
9. **Async programming**: is an overview about how to deal with time-dependent actions and asynchronous computations. We'll talk about promises, futures, generators and the task monad.
10. **Welcome to Fantasy Land**: shows what is Fantasy Land and how it helps different functional libraries to easily interact and work together and shows some libraries that are compliant to this specification.
11. **Hacking the compiler**: gives an overview about Babel and preprocessor tools for JavaScript that allow us to extend the language to reduce boilerplate and make it easier to model our problems. We'll present how to write plugins to the compiler and use

already existing to write better code.
12. **A bit of theory**: is definitely the most theoretical part of the book. We present the roots of functional programming and possibly advanced concepts.
13. **Functional languages targeting JavaScript**: introduces four languages that compile to JavaScript and have a good interoperability with it, but with focus on the functional paradigm. Some are more restrictive than others, however, with restriction comes security and precictability.
14. **Solving real world problems**: presents common daily problems and functional approaches, step by step, to solve them. It is not only about solving problems, but solving problems in such way that they will generate less problems later.
15. **Final considerations**: is our "goodbye". It is not the conclusion, because software development approaches are evolving really fast, and assuming that a specific way to get the job done is the best for all the times is definitely foolish.

Most parts of this book are independent, so you can read chapters in the order you prefer, and the order we set here is only a suggestion (except for the final considerations, of course).

What does "pragmatism" mean?

Pragmatism, noun, a **practical** aproach to problems and affairs. There is no word that can describe this book better. We are going to focus on practical applications of the beautiful and enchanting theory built around functional programming. This book is made for people with basic knowledges on JavaScript programming; you definitely shouldn't have exceptional abilities to understand the things here written: it is made to be practical, simple, concise and after that we except you to be able to write real world applications using the functional paradigm and finally say "I **do** work with functional programming!". This book may be complex for beginners and trivial for advanced programmers. It is designed thinking in the usual programmer that uses JavaScript daily to build any kind of application, but that already **do** have familiarity with JavaScript and is open to the functional mindset. If you are starting to code now, there are better alternatives to this, such as Structure and Interpretation of Computer Programs and How to Design Programs.

When possible, we'll provide the problem that we are trying to solve with alternative implementations and techniques in other paradigms. Programming is made to solve problems, but sometimes solving a problem may generate several other small problems, and this is where we will touch; this is what we will try to avoid. Before continuing, we need to set some premises:

- **Programming languages are not perfect**: seriously. They are built and designed by humans, and humans are not perfect (far from that!). They may contain failures and arbitrarily defined features, however, most times there is a reasonable explanation about

the "workaround".

- **Problem solving may generate other problems**: if you are worried only about "getting shit done", this might be a bit controversial for you. When you solve a problem, have in mind that your solution is not free from introducing problems that other people will have to solve later. Being open to criticism is a good thing here; this is how technology evolves.
- **The right tool for the right job**: this is pragmatism. We pick the tool that solves the problem and introduces the lowest number of side-effects. Good programmers analyse trade-offs. There are a lot of programming languages published and dozens of paradigms, and they are not made/discovered only because "somebody likes writing that way" or "somebody wants to have their name in a programming language" (at least most times), but because new problems arise. It is sensible, for example, to use Erlang on telephone systems, Agda on mathematical proofs and Go on concurrent systems, but it is definitely insane to use Brainfuck on web development and PHP on compiler development!

The author

Marcelo Camargo is a cute brazilian programmer and one of the main evangelists of the paradigm in Brazil. He is the designer of the Quack programming language and colaborator or maintainer of dozens of open-source projects, such as the first unnoficial version of Skype for Linux and improvements on the PHP language. He loves type theory, capybaras and has a cute black cat that is almost always with him while writing. You can find him on GitHub under `/haskellcamargo` or write him a letter on marcelocamargo@linuxmail.org.

The second chance

> And God said, "let there be functions", and there were functions.

In the beginning, computers were very slow, way slower than running Android Studio on your 2GB RAM machine! When the first physical computers appeared and programming languages started to be designed and implemented, there were mainly 2 mindsets:

1. Start from the Von Neumann architecture and **add abstraction**;
2. Start from mathematics and **remove abstraction**.

When it all started, dealing with high levels of abstraction was very hard and costly, memory and storage were really small in power, so imperative programming languages got a lot more visibility than functional ones because imperative languages had a lower level of abstraction

that was way easier to implement and map directly to the hardware. It could not be the best way to design programs and express ideas, but at least it was the faster one to run.

But computers improved a lot, and functional programming finally got its deserved chance! Functional programming is not a new concept. I'm serious, it is really old and came directly from mathematics! The core concepts and the functional computational model came even before physical computers existed. It had its origins on lambda calculus, and it was initally only a formal system developed in the 1930s to investigate computability.

A lot of modern programming languages support well functional programming. Even Java surrended to this and implemented lambda expressions and streams. Most of you program in languages that were created; I want to show you the one which was discovered.

Why functional programming?

If imperative programming and other paradigms, like object orientation, were good enough, why do we need to learn a new way to code? The answer is: survival. Object orientation cannot save us from the cloud monster anymore. More complex problems have arisen and functional programming fits them very well. Functional programming is not "another syntax", as some people think; it is another mindset and a declarative way to solve problems. While in imperative programming you specify steps and how things happen, in declarative (functional and logic) programming you specify what has to be done, and the computer should decide the best way to do that. We've evolved a lot to do the job that a computer can do hundreds of times better than us.

Testability and maintainability

Modular and functional code bases are way easier to test and get a high coverage on unit tests. Things are very well isolated and independent, and by following all the main principles you get composable programs that work well together and have less bugs.

Parallelism

This is really variant with the implementation, but in languages that can track all sort of effects, you get parallelism and the possibility of clustering your program for free.

Optimization

Functional languages tend to be a lot easier to optimize and are more predictable for compilers. Knowing all sort of things that can haven in a program gives the possibility to the compiler to know the semantics of your code before even running it. Haskell can be even

faster than C if you write idiomatic code! The main JavaScript engine, V8, has invested extensivily in otimizations for the functional paradigm.

Less is more

Having a lot of ways to do a job and to solve a problem in the same context is always good, right? Well, not always. If you pluck most programming languages that are largely used by the market, then you have a functional language — and this is valid also to JavaScript. JavaScript has strong ties with Scheme and has the perfect potential to be a very good functional language. Remove loops, mutability, references, conditional statements, exceptions, labels, blocks and you virtually have a dynamically typed OCaml dialect. I'm serious, you can even define functions in JavaScript without giving importance to the order they occur, just like OCaml em Haskell do — this is what we can call hoisting and we'll be seing how to take advantage of this in the next chapters.

Don't be miser. Do you really need all that different unaddressed language constructs? In the chapters of this book we'll discard a large part of JavaScript and we'll focus only in an expressive subset: functions and bindings, it is everything we'll need for now! Before learning functional programming with JavaScript, you first will have to think about unlearning some things. This is necessary to avoid language vices and to estimulate your brain to solve problems in a declarative way. For now, you'll have to trust, but in the course of this book you'll see how everything makes sense and fits into the purpose.

Tooling

We'll focus more on program semantics and correctness than execution, but if you really want to learn, only reading this book will not be enough.

Roots of the evil

Null considered harmful

And by `null`, we also mean `undefined`. It is hard to see a programmer who never had any sort of problems with null references, null pointers or anything that can represent the absence of a value. For JavaScript programmers, `undefined is not a function`, for the C# young men in suits, `NullReferenceException`, and for unlucky one who use need to use Java, the classic `java.lang.NullPointerException`. Do you think we really need `null` as a special language construct? In this point, there is also a consensus between functional programmers and object-oriented programmers: `null` was a secular mistake.

A brief history

Null references were introduced in ALGOL-W programming language decades ago; 1964 to be specific. They hadn't any special purpose except they were considered "easy to implement" by the language designers, and a lot of modern languages implemented it for the same reason — and it gone viral. At that time, most of the softwares, including the compiler, were written in machine code with little or no abstraction. But what is wrong with `null`?

(too many) runtime exceptions

I don't mean runtime exceptions would be abolished if `null` were, but they would reduce a lot for sure. Unexpected exceptions due no null references are very common in the life of Java, JavaScript, C, Go and C# programmers — and we could avoid that with specific techniques. We can't have exceptions involving `null` values if we don't have `null` values.

It increases complexity

When you have to compare for null to avoid runtime errors, your code logic forks and creates a new path of possibilities. If you use lots of `null` values, the chance of having spaghetti code that becomes hard to maintain is really big. Object-oriented languages recommend the null object pattern, while functional languages use monads for that. Luckily, there are lots of libraries and monadic implementations for JavaScript to abstract and avoid this problem. We'll see monads in details and how to properly handle `null` later — for now we'll be presenting the problems and seeking the solution will be a task for the next chapters.

Unpredictable types

Why would you return `null` in a function that retrieves the list of users of the database when I can just return the **empty representation** of the expected type? — an empty array in this situation. The fact is that `null` can inhabit **any** type, and this makes the task of debugging a lot harder. Which one is more pragmatic?

```
const showUsers = () => {
    const users = getUsers()
    if (users !== null) {
        users.forEach(console.log)
    }
}
```

```
const showUsers = () => getUsers().forEach(console.log)
```

Don't make it complex when you can make it simple.

True, false ...and null!?

Unless you have a **very** good excuse, don't return `null` when all you need to do is giving a simple boolean answer. First thing: `null` is not `false`, `null` is the absence of value, and in this case we don't have a boolean representation, but a three-state model. Booleans should never be nullable (or they aren't booleans at all) — and `null` is not the same as `false` and not even supposed to be.

Mutability can kill your hamster

One of the pillars of functional programming is immutability, and changing things is a very common thing in imperative and object-oriented programming languages, but how one can model computations and problems without even being allowed to have variables?

Mutability is about showing the computer **how** to solve your problem and not presenting out declaratively **what** is the problem that it should solve.

Before presenting how we can do it, let's first examine why mutability is evil and you should avoid it when possible and sensible if you care about the life and health of your hamster. This is definitely not a rant about mutability in general, but an overview of the problems that are undeniably common in real world projects. There are situations where mutability is necessary, but they are the exception, not the rule.

Mutable data is inherently complex

And mutability without explicit control is what makes software development difficult. The control flow of your program takes more possible paths and more things to worry about. When maintaining code bases with lots of mutable or global variables and references, there are not warranties that touching and changing a function will not present side-effects in other part of the program that may be completely unrelated. Functions with only input and output give you warranty that they will not affect other parts of the program and are predictable. Worrying about program safety should be a work of the compiler, not yours. The programmer should worry about problem modeling and the only errors with which they should worry about should be from business logic, but most mainstream languages transfer their work to the depressive programmer.

Multithreading is hard

If you believe that imperative languagens and things which are closer to Assembly are better for distributed systems, you are wrong, and increasingly wrong. Immutable and predictable code is inherently easier to optimize to run in multiple cores and threads because the compiler knows that it can safely run different parts of the program independently knowning that there is no interdependence among them. A function that has some input and aways the same output and that doesn't touches parts of the program it shouldn't or emit side-effects is always thread-safe and optimizable.

Another problem with mutable state is when you deal with server-side development with a stateless server. Running your application in clusters (multiple cores or machines) with a stateful server becomes a major source of headache because the mutable state of the server is not shared across all the possible instances of the application. The absence of a global mutable state is what allows Haskell to be so optimizable. Erlang and Elixir store and share this state using the process calculus (like π-calculus).

Loss of reversibility

Reversibility matters a lot. Since 2015, new tools for front-end development have emerged and a large part of them is about state control and reversibility, such as React and Vue. Reversibility is about having the control of your program and being able to replicate the point of a program based purely on it's current state. This also makes debugging and finding errors a lot easier because not reproducing errors can no longer be an excuse. When you have reversibility and state control, complexes things become trivial, such as restoring the program state on restarting it or implementing "undo" and "redo" operations.

Debugging can be tiresome

And I'm not joking. Debugging becomes still harder when you are dealing with multiple processes and asynchronous programming. Did you ever wonder how in the world that variable suddenly had its value changed? When you combine mutability with nullity, you create a dangerous monster.

References may be a problem

And the problem increases when you add mutable references; they are, at first, hard for compiler optimization and creates a snowball where any point of the application that receives a reference is able to mutate the value hold by the variable. References are very useful in some situations, but unless you really need to use them and you know what you are doing, avoid them JavaScript has a special problem where some methods work on references instead of values, and you can easily lose the control of the situation:

```
const list = [8, 3, 1, 2, 7, 4]
// The sorted list
console.log(list.sort()) // [1, 2, 3, 4, 7, 8]

// Then original list... oh wait!
console.log(list) // [1, 2, 3, 4, 7, 8]
```

So, if in any point of the application a function that is called to work with the list decides to sort the values, it will mess up with your original data and increase the possibility of a wrong or unexpected computation. Some functions that operate on arrays to keep distance:

- `copyWithin`
- `fill`
- `pop`
- `push`
- `reverse`
- `shift`
- `sort`
- `splice`
- `unshift`

They are documented in the Mozilla Developer Network in the section "Mutator methods" of the `Array` object.

Our computers are powerful enough to deal with immutable data transformation without major overheads, that are lots of libraries that do that (but you might not need them), so there is almost no reason to be concerned with micro optimizations that will already occur in compile time. Yes, compile time. The most famous implementations of JavaScript, such as

the one which runs on Chrome and Node are compiled (and not directly interpreted) to native machine code right when the program is "started", this is known as JIT — Just in time compilation, and then the compiled program executes.

Type coercion is a bad boy

While static typing helps avoiding mistakes and unexpected behaviors, type coercion helps losing the control about your data values and types. Type coercion means that when the operands of an operator have different types, then they are converted to an equivalent and acceptable value by that operator. For instance, when you do:

```
1 == true
```

This is `true` only because, as longer as there is no comparison for different types, `true` on the right side is converted to a number (based on `1` in the left side), in this case, `true` becomes `1`. The operator `===` prevents this conversion from happening, and it is what we will be using from this point.

The type system of a programming language defines what types of data can exist in the language and how operations can be applied to them, and this sort of thing exist to prevent the programmer from shooting theirself on the foot. Type coercion is about **implicitly** converting incompatible values when there is no valid operation for that type of value.

In languages like Python, even with dynamic typing, operations like `"foo" + 1` will emit a runtime error:

```
>>> "foo" + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

And this helps preventing a lot of bugs that would be caught only in distant point of the program. Type coercion is a misnomer, as long as values have no types in the sense of static typed languages, but it is a definition of how different values interact. The ECMAScript specification about type coercion is really big; there are lots of rules and exceptions and trusting on these conversions is a fatal error.

This is a major source of bugs in JavaScript programs because there is no **sensible** nor **consistent** specification of how these values should interoperate and behave, but empirical and variant ones, or do you expect logically that `[] + {}` is `[object Object]` and `{} + []` is `0`? I don't think so.

Somebody stop this!

There are a lot of articles about the `this` keyword in JavaScript, and **this** was one of the worst design issues of the language. Some people argue that there would be names that express better the concept of `this`, like `potatoes`. Even JavaScript programmers with years of experience get confused about it and it is a common source of mistakes among them. There are several different ways to set the value of `this` inside a function, such as calling a method from an object, calling a function alone, using `call`, `apply` or even using `new` to create an instance. It's really a handyman keyword. Telling the behavior of a program piece only looking at it when it has `this` is not as easy as it is with pure functions because you need to worry and care about the **context** of call of that function. It also has different behaviors when using `function` keyword and `() =>`, the fat-arrow, which captures the top context instead of creating another, making the use of the two forms of declarations of anonymous functions not interchangeable.

Behaviors of this

Method call

```
const person = {
    fullName: 'Mr. Pickles',
    sayName: function () {
        console.log(`Hello, my name is ${this.fullName}!`)
    }
}

person.sayName() // Hello, my name is Mr. Pickles!
```

Note that using `() =>` instead of `function` would capture the top `this`, making `this.fullName` be `undefined` in this situation.

Meet ESLint

Using a linter is a nice way to help you avoiding mistakes and encouraging a standard coding style. There is an excelent tool called ESLint for that. We will be using it in these series to restrict some language features while encouraging others.

Restricting imperative syntax

We will configure the linter to restrict imperative syntax to avoid relapses. The mindset used in functional programming is different, simpler than imperative, so we can eliminate language features that would cause confusion and that have good functional replacements.

Types

JavaScript by default doesn't provide support for static type or even runtime type checking. By having type coercion, all the syntactic valid programs are accepted and type errors will only be caught in runtime. For the sake of sanity, there are good tools we can use to make JavaScript code safer and identify most of the runtime errors of a program before they happen, I mean at compile time, like Flow or even TypeScript, the JavaScript superset.

Why types matter

Do we really need types? At first, this should be an obligation of the compiler; it should be its responsability to ensure that your program is correct and let you worry only about the logic, but this is not what happens on the most popular programming languages. You might not need static type checking, but, for sure, having it will help you writing programs that are more correct and safer, avoiding a lot of possible errors that would only happen in specific situations when executing your program. In fact, static type checking will not replace unit tests at all, but will replace the sort of them that are applied to inputs of invalid types.

When talking about type checking, most people remember about Java, but this is definitely a terrible example of how a type system should be. Seriously, the type system implemented in Java forces you to declare things that the compiler already knows and accepts invalid programs because it is not well designed to deal with a thing we will call subtyping. In the section 12, "A bit of theory", we will give you a gentle introduction to type theory; we will not focus specifically on Java, but in systems that can be implemented in the subset of functional languages and explain how type inference works and why most of the languages have awful type systems that make a lot of people hate static typing. If you say you hate static typing, I'm 99% sure you don't hate static typing, but having to write type signature for obvious things and having to waste timing making the work that the compiler should do.

Flow is your friend

Hopefully, to save us from the world of badly-typed programs, we have Flow.

Monads, monoids and functors

I'm pretty you sure you already have heard about the term "monad". A lot of people talk about them, but a few know what they really mean and how they can fit into functional programming. We'll be focusing on the practical approach of monads, but give a gentle introduction to the theory. The most advanced theory can still be found in the chapter 12 of this book.

Welcome to Fantasy Land

Unicorns and rainbows

Isn't that cute and wonderful? Fantasy Land is a specification for algebraic JavaScript, that is, the specification of how algebraic structures should be interoperable in the JavaScript environment. This means most part of JavaScript libraries and tools with focus in functional programming will follow these laws and specification, making them easily interoperable. This is the theory that will lead to a pragmatic and standardized practic.

But first, how can we define an algebra?

- **A set of values**: like primitive numbers, strings or even other data structures.
- **A set of operators**: that can be applied to values.
- **A set of laws**: the operations applied to these values must obey.

Each Fantasy Land algebra is a separate specification. Some algebras may depend on other algebras. The JavaScript primitives also have equivalents to some of these algebras, as we will see in some occasions. The specification warrants that values that satisfy to specific algebras can be safely replaced by other values that also respect them. For example, we can have:

- One list can be replaced by another if they have the same length.
- Two objects have the same keys and their values have the same or compatible types.
- Two functions have equivalent output types for equivalent input types.

Understanding type signatures

We'll be using a notation similar to Haskell type signatures to express the types that the algebras must satisfy, where:

- `::` means "is a member of".
  - `e :: t` is the same as "the expression `e` is member of type `t`".
  - `true :: Boolean` means that "`true` is a member of type `Boolean`".
  - `42 :: Integer, Number` means that "`42` is a member of type `Integer` and type `Number`".
- New types can be created via type constructors.
  - Where type constructors can take zero or more parameters.
  - In Haskell, `True` is a type constructor that yields a `Bool` and takes no parameters.
  - `Array` is a type constructor that takes one parameter.
  - `Array Number` is the type of any array that holds only numbers, like `[1, 2]`.

    - Types can be nested. `Array (Array Boolean))` is the type of all arrays that have arrays of booleans, like `[[true], [false, true], []]`, where `[]` can be seen as an array compatible to any type parameter.
- Lowercase letters stand for type variables:
    - Type variables can take any type unless they have constraint restrictions, as we'll see bellow.
    - `[]` can be seen as `Array a` for example, where `a` can be `String`, `Number` or any other type.
- `->`, the arrow, stands for function type constructor.
    - `->` is an infix binary type constructor that takes the input type on the left and output type on the right.
    - The left type of `->` can be a grouping for functions that take more than one parameter, for example, `(Number, Number) -> Number` is a function that receives two numbers and returns another.
    - Parentheses on the left of `->` are optional for unary functions and obligatory for the rest.
    - Functions that take no arguments can be represented by the unit type `()`.
    - `a -> a` is a function that receives a parameter and yields a value of the same type.
    - `Array String -> Array Number` is a function that receives an array of strings and yields an array of numbers.
    - `(String, String) -> a` is a function that receives two strings and yields anything.
    - `() -> ()` is a function that takes no parameters and yields no results.
- `~>`, the squiggly arrow, stands for method type constructor.
    - If a function is a property of an object, then it is a method. Methods have implicit type parameters that is the type of the object that holds the property.
    - `Object ~> () ~> String` is the signature satisfies by a method that, when applied to the object and without parameters, yields a string, such as `Object.prototype.toString`: `(1).toString`.
- `=>`, the fat arrow, stands for constraints in type variables.
    - `Monoid a => a ~> a -> a` stands for a method applied to an object of type `a` that receives another instance of `a` and yields also an `a` where all these values satisfy to the `Monoid` laws.

Functional languages targeting JavaScript

If you are still not convinced about using JavaScript as a functional language, but enjoys the environment it provides and/or needs to do web or server development with it, then there are several alternative languages that encourages functional programming better than JavaScript and compile to it. You may think, if it compiles to JavaScript, why not writing JavaScript? Because languages provide abstractions to make problem solving easier. This would be the same thing about arguing about writing Assembly because "languages compile to it". There are almost as many languages that compile to JavaScript as there are JavaScript frameworks and libraries.

LiveScript

CoffeeScript is for Ruby as LiveScript is for Haskell. LiveScript is a functional language with strong roots on Haskell and F# with dynamic type checking that compiles to pure and mappable JavaScript. It's very useful for prototyping and data transformation. It provides a standard library called `prelude-ls` with lots of functions to handle lists, objects and other data structures with focus on composition and immutability, but still allows everything that JavaScript does, such as loops and random effects (not that you should use them). LiveScript is a fork of Coco language, an indirect dialect of CoffeeScript, but it is a lot more expressive. It also provides source-maps to allow debugging on VSCode, Chrome or Firefox without headache or suffering.