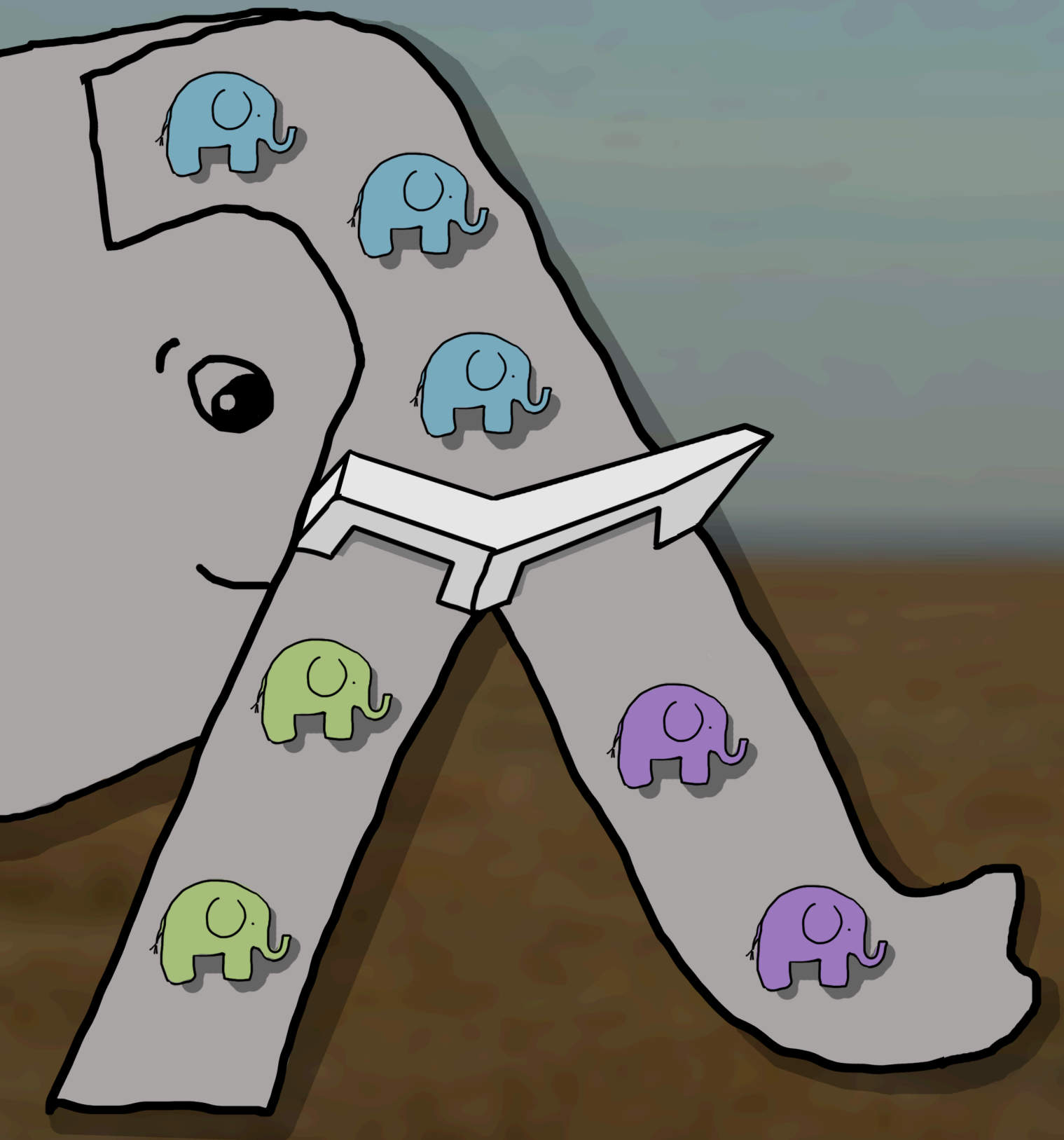
# Thinking Functionally in PHP

By Larry Garfield



# Thinking Functionally in PHP

Larry Garfield

This book is for sale at http://leanpub.com/thinking-functionally-in-php

This version was published on 2021-04-25

# Also By Larry Garfield

A Major Event in PHP

Exploring PHP 8.0

# Contents

# Preface

Functional programming has been an interest of mine for years. That's not to say I've been an expert in it, or that I've done much work in strictly functional languages, just that I've been interested in the topic.

Sadly, I've long found the available documentation and tutorials on the web to be lack-luster. Most of them either cover the very basics and then stop, or assume you're already an expert in Haskell and so explain advanced topics in the most convoluted and Haskell-centric way possible. Neither of these did I find useful.

I have also found that the Curse of Monads seems to be true, as nearly every explanation I found about that mythical concept was obtuse, assumed you had three years of advanced mathematical training, or so dumbed-down as to be actively harmful to understanding.

> "Once you understand monads, you immediately become incapable of explaining them to anyone else."
>
> –Gilad Bracha

Combined with PHP's overall clunky syntax for doing functional-esque code, I generally didn't go further than "pure functions are your friend," either in my own code or what I explained to others.

That is, until PHP 7.4.

PHP 7.4's introduction of short lambdas is, as we'll see in this book, a game-changer. While it doesn't make anything new *possible*, it makes a lot of things suddenly *practical*. That makes all the difference, so I decided it was time to buckle down and really dig into functional programming. And what better way to learn than to force yourself to teach it to others?

I don't mean that sarcastically; in fact, that's been my history with functional programming from the beginning. I submitted a talk to a conference back in 2012 on functional programming before I really knew what I was doing, because that gave me a reason to learn it well enough to be able to explain it. I went on to give it 10 times. This book has had the same effect.

I knew when I started out that I was going to have to actually learn, finally, what all this category theory fuss was about. I did not expect it to be as hard as it was; apparently the Curse of Monads applies to most of category theory. A large part of the issue seems to be that no one can agree on how to call anything; it took reading a dozen tutorials in a half dozen languages to realize just how much the terminology varied, and just how much of a problem that causes for people trying to learn functional programming, category theory, or anything else.

I never would have thought that software engineering could be held up as an example of having consistent terminology, but I suppose all things are relative.

The central crux of this book is Part Two, in which I tackle the hard, academic, theoretical bits. It seems to be a trope that any time someone thinks they have understood monads they write an online tutorial about it, and I suppose I am no different. It is my hope that by writing that section, and rewriting it multiple times, while learning the material itself I was able to straddle the threshold between this world and the next (the one in which one does not understand monads, and the one in which one cannot explain them) long enough to capture a reasonable explanation of them. Should you find that I was successful, I would love to hear about it.

I am indebted to a number of people for their help in making this book happen.

- Nash van Gool, who provided early validation that my early efforts were not entirely incomprehensible.
- Nash and my colleague Chad Carlson, who served as beta testers for the first draft of the manuscript.
- Shayna Steinberg, for reviewing the category theory section and validating it is comprehensible even for non-programmers/mathematicians.
- Bartosz Milewski, whose book and website "Category Theory for Programmers[1]" is the best tome on the subject to date. He also was kind enough to answer my random validation questions on online forums, even though he had no idea why I was asking.
- My editor, Kara Ferguson[2], without whom it's quite possible there would be not a single comma in the right place in this entire book.
- Rebekah Simensen[3], who provided the delightful cover art you have probably already enjoyed. (Assuming you looked at the cover before opening this book, anyway, which seems likely.)

And of course to you, dear reader, as a book that is not read may as well not exist. It is my sincere hope that you find this volume educational, entertaining, and eye-opening. Or at least that it serves you as more than a monitor stand.

May your life be as pure as your code. I wish you well.

–Larry Garfield, @Crell

[1] https://bartoszmilewski.com/2014/10/28/category-theory-for-programmers-the-preface/
[2] http://www.karaferguson.net/
[3] https://ninjagrl.com/

# Introduction

In recent years, functional programming has become increasingly popular as a concept. Even in "non-functional" programming languages, the concepts and practices of functional programming have become more mainstream.

Why is that? Are they actually useful, or is functional programming in a language other than Haskell just trying to fit a square peg into a round hole? What even is functional programming, and does it make sense in PHP?

This book aims to answer those questions. (Spoiler alert: The answer is yes.) While writing PHP as if it were Haskell, ML, or LISP is not going to produce a good result, the underlying principles behind functional programming offer many advantages, even in PHP. Other parts, however, do not, and that's okay.

The aim of this book is not to convince you to write all PHP code in a strictly and rigidly functional fashion. Rather, the aim is to encourage you to approach problems from a functional mindset and apply the underlying principles of functional programming… most of the time. Many of them, in fact, you have no doubt encountered before in other contexts talking about "good code" under different names, such as the SOLID principles. That's a good sign the concepts involved are a good idea.

## Functional vs. Procedural

There are many different paradigms for conceptualizing computation and programming, but the two oldest and most widely applicable are *procedural programming* and *functional programming*. Both were developed in the 1930s very close together, and have been shown to be equivalent; that is, there is no programming problem that can be solved in a procedural approach that cannot also be solved in a functional approach, and vice versa. One paradigm may be easier to write, or easier to read, or easier to think about, or more performant, or require fewer lines of code. Which one has the "edge" varies widely from one situation to another, but if it's possible to solve in one, then it's possible to solve in the other.

One could argue that since both are equally capable, a programmer needs to learn only one to solve any solvable problem. That would be a short-sighted view. By that logic, one would need to learn only a single Turing-complete language and be done with it, since all Turing-complete languages are, by definition, equally capable.

While it's trite to fall back on cliches like "use the right tool for the job," it's a cliche because it's true. But being able to determine the right tool for the job requires being at least competent in multiple tools. In the case of programming, that means being at least competent in multiple programming

languages as well as multiple programming paradigms, if only to know when to *not* use a particular one.

Most programming resources, however, are geared toward a procedural mindset. That's not surprising. It is the approach easiest to explain to a new developer, and at the end of the day, nearly all computer hardware in existence is procedural; anything else is simply syntactic sugar on top of assembly. That is unfortunate, though, as procedural is also the paradigm that scales least-well to large-scale systems, concurrency, and the ugly complex mess that is 21st-century computing.

What functional programming and procedural programming share, at least in their modern forms, is the division of programs into separate logical chunks that can be mixed and matched and reused. Even if the syntax is often similar or identical, however, functional programming goes a step further and makes those logical chunks first-class citizens that can be dynamically created, modified, passed around, even stored and reused. We'll see concrete examples of that in Part One.

## Object-Oriented Code

The dominant programming paradigm today is neither functional nor procedural, but *object-oriented programming* (OOP)—at least, that's the conventional wisdom. One issue with said conventional wisdom is OOP is a very loosey-goosey paradigm, with several sub-paradigms that actually get used. Far and away, the most common is what I term "Classic OOP," as in, OOP that's based in classes. C++, Java, C#, and PHP are all in this family, and with the introduction of explicit classes in ECMAScript 6 JavaScript sort of is.

There are other variants of OOP, however, some of which do not even include the quintessential "inheritance" concept. In practice, most OOP code in the wild is what I would term "procedural OOP," that is, procedural thinking wrapped up in OOP style. That is not necessarily a bad thing; it is the dominant model today, so it must be doing something right. However, it is not fully message-centric OOP (a la Smalltalk). It also means there really is such a thing as "functional OOP," or "object-functional" code.

Contrary to the Reddit consensus, OOP and functional programming are not adversaries that force you to choose One True Code Paradigm. Rather, we can and should leverage the lessons of both to produce the best solution to the problem.

## Functional Style? Functional Language?

One difficulty in understanding functional programming is that, much like object-oriented programming, there is no one clear definition of functional programming. On one level, it's simply "programming with functions the way math does," which is a nice and uninteresting definition. However, doing so quickly leads down a rabbit hole of category theory, monoids, endofunctors, generic type systems, pathological recursion, and other weird sounding words and phrases that are prone to turn off even the most receptive audience. In many cases it ends up looking like an

unintentional Mott-and-Bailey fallacy[4], an informal logical fallacy[5] where one argues two related points at the same time, one controversial and one not, and switches between the two depending on which is more advantageous in the moment.

There is, really, no universal definition of functional programming, nor of a functional programming language. If "functional programming language" means "a language in which one can do functional programming," then nearly all modern languages are functional; this is not a helpful definition.

A better definition is to distinguish *functional-style programming* from a *functional language*.

- *Functional-style programming* is a particular way of approaching a logical problem in code. It includes a couple of fairly simple core concepts, plus a huge array of concepts that naturally flow from those. Most of those are not all-or-nothing, making it possible to write "functional-ish" code in almost any modern language even if the syntax may end up a bit clunky.
- A *functional language* is a language whose syntax and semantics are geared toward making functional-style programming easier and more natural, often (though not always) at the expense of other paradigms. Often those functional concepts are baked-in assumptions in the language, and if you are not aware of those assumptions, you're likely to get bitten.
- A *strictly functional language* is one whose syntax and semantics are geared toward writing in a functional style only, period. These languages go all the way down the rabbit hole and make thinking about a problem any other way frequently impossible. They also have a reputation for being impenetrable, precisely because they only support functional models.

PHP, in particular, is a language optimized for procedural OOP; it supports classic OOP, and that is the code style most widely used, but the single-threaded, shared-nothing architecture is intrinsically procedural. Since PHP 5.3, however, it has had syntactic support for functional-style programming. PHP 7.4's introduction of short-lambdas further simplifies it far enough that functional-style code is now entirely viable.

That said, would I advocate writing 100% purely functional code in PHP? Not really. Many of functional code's advantages do not actually apply in a single-threaded, shared-nothing environment, and others will work but collide with other limitations of the language to make them less friendly to work with.

## Type Systems

Type systems are a tricky topic in functional programming. On the one hand, type systems are entirely orthogonal and unrelated to functional programming. Some functional languages have extremely robust type systems (e.g., Haskell), while others have seemingly none at all (LISP). Yet, they're often talked about in the context of functional programming. Why?

[4] https://rationalwiki.org/wiki/Motte_and_bailey
[5] https://en.wikipedia.org/wiki/List_of_fallacies#Informal_fallacies

The answer lies in the previous observation that functional languages tend to have either no type system or a very robust one. Rarely have I seen a mediocre functional type system. While functional programming itself does not in any way require having an explicit type system, in practice functional code encourages and requires a very robust type system. A weak type system often cannot handle the complexity that a functional code base necessitates.

PHP is unusual in this case in that it has a relatively weak type system (compared to what languages like Rust or Haskell or even modern Java offer), but it's all opt-in. As we'll see, the result is that taking a fully functional approach will, in some cases, lose us some explicit typing ability. I don't see that as a fatal problem, but rather as an indication of what sort of type system improvements PHP should prioritize in the future.

We'll cover some typing concepts along the way, as necessary, but this is not primarily a book about typing.

## What to Expect

This book is divided into four parts:

- In Part One, we will examine the fundamental principles of functional programming as a concept, using PHP as the example language. The results will not always be syntactically pretty, but that's not the goal; understanding the value of the principles is, and we'll use concrete PHP examples to go a little way down the rabbit hole.
- In Part Two, we'll step back and look at the theory that underpins much functional programming, specifically category theory. This is not a book on category theory, but a cursory understanding of the basics will greatly help to explain how functional programming and other formalisms really do make for more robust code.
- In Part Three, we'll bring the theory and practice together to demonstrate some more advanced techniques that make the theory concrete and make for more robust code at the same time.
- In part Four, we'll step back and consider the broader picture of what else functional programming could be, and where it should sit in your toolbox.

# Part One

# Pure Functions

The first and most crucial aspect of functional programming is the idea of a "pure function." What makes a function pure? The fact that it's an actual function and not something else.

If that seems circular, let's back up a moment. One of the key challenges in discussing functional programming is software engineering and computer science are, in fact, two separate fields that have evolved in parallel with each other. They interact frequently, but not consistently. As a result, a great many concepts have a different name in software engineering than they do in computer science, and many others have the same word in both disciplines that mean different things. In order to establish a common vocabulary, therefore, let's start at the very beginning.

In nearly all programming languages, there is an idea of a block of code that is reusable over and over again in multiple places, often with slight variation. The term for such a block is a *procedure*, or *sub-routine*. A procedure is any syntactically reusable block of code that can be referenced and, optionally, passed some parameters or arguments.

In mathematics, a *function* is a black box that is given inputs and reliably returns an output. How it does so is irrelevant; it could do some computation, or it could simply have a giant internal lookup table. However, the key term there is "reliably"; the same inputs will always produce the same outputs, guaranteed, and nothing else will change about the state of the universe except that a value is returned. Two inputs may result in the same output, but the same input can never, ever produce different output.

A function in mathematics is not a thing that happens but a *relationship* that defines the map from some input to some output.

In almost every mainstream modern language, the concept of a procedure and a function are folded together into a single concept termed "function." In PHP-speak, both of the following are "functions":

```
function output(string $message): void
{
    static $n = 1;
    global $preamble;
    echo "The {$n}th message is: {$preamble} {$message}". PHP_EOL;
    $n++;
}

function add(int $a, int $b): int
{
    return $a + $b;
}
```

Both are procedures; however, only the second is a "function" as mathematicians would define it. The `output()` function takes a stealth input (a global variable), it produces a stealth output (it prints a message), and it has a different effect if called a second time with the same input (because it retains internal state via a static variable). All of those attributes violate what we expect of a "function" in the mathematical sense.

Of course, modern languages confuse things by using the term "function" willy nilly. Therefore, what mathematics calls a "function" programming instead calls a "pure function."

# Define "Pure"

What makes a function pure? A procedure is a pure function if it follows two restrictions:

1. It is *idempotent*. That is, the same explicit inputs are guaranteed to always produce the same outputs.
2. It has no side-effects. That is, there is no effect on any value or data stream other than the return value being returned.

These rules are very simple and usually easy to achieve, but very powerful. By restricting ourselves in this way, we buy a number of assumptions.

1. A pure function is incredibly easy to unit test. There is essentially no set up and no mocking necessary; just call it with known parameters and see if the result is what you expect.
2. It has *referential transparency*. That is the fancy way of saying a function is equivalent to its return value. As long as `add()` is pure, then replacing all instances of `add(5, 5)` with `10` is guaranteed to not change the result of the program.
3. Because we know that the same inputs produce the same output and because it's referentially transparent, it is trivially easy to cache. Caching is, in essence, replacing subsequent function calls with a previously computed value.
4. Conversely, while calling a pure function multiple times with the same input may be wasteful, it cannot be a source of bugs. That's assuming you meant to call it with the same value multiple times; if not, the bug is in your parameters, not the function.

As we'll see in the next chapter, those assumptions allow us to do a great deal more once we are dealing with higher-order functions. Stay tuned!

# Composing Functions

Another important attribute is that composing pure functions always produces a pure function, but calling an impure function from a pure function makes the pure function impure. That is:

```php
function add(int $a, int $b): int
{
    return $a + $b;
}

function subtract(int $a, int $b): int
{
    return $a - $b;
}

function compute(int $a, int $b, int $c): int
{
    return subtract(add($a, $b), $c);
}

function output(string $message): void
{
    echo "The message is: {$message}". PHP_EOL;
}

function run()
{
    $val = compute(30, 10, 8);
    output("The result is: $val");
}
```

`add()` and `subtract()` are pure functions, because they obey the two rules of pure functions. This means `compute()` is also a pure function, because it obeys the same rules and only calls other pure functions. However, `output()` is not a pure function because it has a side effect of producing output. `run()` calls both a pure and an impure function, and is thus impure. We cannot make any such assumptions about it.

Functional programming is built on pure functions. In strictly functional languages, writing a not-pure function is hard and requires extra syntax to remind you that you're doing something dirty. PHP is not a strictly functional language, so it's on us as developers to self-check and ensure our functions are pure whenever possible.

In practice, ensuring you have pure functions is straightforward:

1. Avoid global values. Always. Period.
2. Avoid static values in functions. Pretend they do not exist.
3. Avoid passing or returning values by reference. In practice, there are extremely few use cases for that anyway in PHP, as PHP automatically supports copy-on-write making passing large,

read-only parameters very memory efficient. This becomes a bit trickier when passing around objects, but there are ways to handle that which we will see later.

4. If you need to do IO (read from a database, print to a screen, make an HTTP request, etc.), have a function (procedure) that does just that and nothing else, and returns its value. That function is impure, as is its caller, but other functions called by its caller need not be.

For example, this function reads a value from a database and then does some processing on it:

```
function get_user(Connection $db, int $id): array
{
    $result = $db->query("SELECT * FROM users WHERE uid=:uid", [':uid' => $id]);
    $row = $result->fetchRow();

    if ($row == 'false') {
        return [];
    }
    if ($row['expired'] == 1) {
      return [];
    }

    return $row;
}
```

Whereas a more robust approach would split the non-fetch functionality off to its own routine:

```
function fetch_user(Connection $db, int $id): array
{
    $result = $db->query("SELECT * FROM users WHERE uid=:uid", [':uid' => $id]);
    return $result->fetchRow();
}

function validate_result($row): bool {
    if ($row == false) {
        return false;
    }
    if ($row['expired'] == 1) {
      return false;
    }
    return true;
}

function get_user(int $id)
```

```
{
    // ...
    $user = fetch_user($db, $id);
    if (validate_result($user)) {
      return $user;
    }
    else {
        // Some kind of error handling.
    }
}
```

`validate_result()` is now a pure function, and thus far more predictable and easier to test.

Because most of functional programming is based on pure functions, for the rest of this book, whenever we talk about "functions" assume we mean pure functions unless explicitly stated otherwise.

## Binary Functions

A particular subset of functions are *binary functions*. Binary functions are functions which take two parameters of the same type and return an item of that type. That is, the following is a binary function:

```
function multiply(int $a, int $b): int
{
    return $a * $b;
}
```

But this is not, because it does not return the same type:

```
function greater_than(int $a, int $b): bool
{
    return $a > $b;
}
```

So far, that's not all that interesting. However, there's another subset of binary functions that are; specifically, binary functions can be *associative* and have an *identity element*.

A binary function is associative if the order of grouping doesn't matter when chaining multiple calls together. Let's look at multiplication again, this time with more traditional syntax.

```
$a * $b * $c * $d
```

Multiplying `$a` by `$b`, then by `$c`, then by `$d` produces the same result as multiplying `$a` by `$b`, `$c` by `$d`, and then those results together. Or, more visually:

```
$a * $b * $c * $d == ($a * $b) * ($c * $d)
```

An identity element is also known as a neutral element and is essentially the "no op" value. For multiplication over integers, the no-op value is 1. That is:

```
$a * 1 == $a
```

For every possible integer value of `$a`.

It turns out this combination of properties is rather useful in general. So useful that it has a fancy name: a *monoid*.

A "monoid" refers to a function that:

- Is pure
- Is a binary function (takes two parameters and returns one value, all of the same type)
- Is associative
- Has an identity value

More formally, we can say that multiplication is a monoid over integers. Stated more precisely, there is a monoid <$\mathbb{Z}$, **multiply**, **1**>: a set of values (integers), a binary operation (multiply), and an identity value (1). Multiplication may or may not be a monoid over other types of values, and for some types of values (e.g., turnips, employees, or shopping carts), it doesn't even make sense.

While multiplication is fairly basic, the concept of a monoid applies to all sorts of types and operations. Consider list concatenation, for instance:

```
function concat(array $a, array $b): array {}
```

Is that a monoid? Let's check the rules:

- Is it pure? Check.
- Is it binary? Check.
- Is there an identity value? Yes, an empty array.
- Is it associative?

We could go through a formal proof of associativity, or write tests to demonstrate it, but for now, let's just logic it through. Given lists `[1, 3]`, `[5, 7]`, and `[9, 11]`, is the following true?

```
concat(concat([1, 3], [5, 7]), [9, 11])) == concat([1, 3], concat([5, 7], [9, 11]))
```

Yep, it is. Both give the same result, `[1, 3, 5, 7, 9, 11]`. For now we'll go with that intuition and conclude it is associative (spoiler alert: It really is.), so we can conclude concatenation is a monoid over lists; or, more formally, **<lists**, **concat**, **[]>** is a monoid.

This concept will become more important later when we discuss value objects.

# First Class Functions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Anonymous Functions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Closure Objects

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Short Lambdas

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Object Binding

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

# Memoization

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

# Currying

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

# Mapping

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

# Filter

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

# Reducing

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

# Concatenation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

# Recursion

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Recursion

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

# Immutable Value Objects

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## What PHP Does

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Making It Immutable

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Objects, Functionally

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

### Service Objects

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

### Value Objects

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Evolvable Objects

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

# Dependencies

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

# Entity Objects

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Ignore It

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Mutation Instructions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Event Sourcing

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Functional State Management

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

# Intra-Function Immutability

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

# Part Two

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

# Category Theory

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## What Is a Category?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Objects

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Some Definitions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Structure

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Let's Get Meta

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Isomorphism

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

# Monoids

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

# Categories in Code

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Converting Structure

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Function Objects

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

# Fun With Functors

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Combining Functors

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Monads in Programming

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Anatomy of a PHP Monad

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

# Algebraic Data Types

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Products

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Product Types in PHP

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Coproducts

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Coproducts in PHP

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

# Part Three

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

# Handling Null



## The Problem



## The Example



## The Category Theory Solution



## The Programming Solution



### Unit



### Bind

### Getting Back Out

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## An Alternate Implementation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## A Note on Types

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

# Either/Or and Error Handling



## The Problem



## The Example



## The Category Theory Solution



## The Programming Solution



### Railway Programming



### More Pathways

### Collapse the Types

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

### Alternate Approach

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Either Either?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

# Tracking Data

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## The Problem

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## The Example

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## The Category Theory Solution

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## The Programming Solution

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

# Combining Monadic Behavior



## The Example



## The General Approach



### Tokens



### States



## Binding



## The Lexers

## Testing

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Analysis

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Being Lazy

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

# Part Four

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

# Features of Functional Languages

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Auto-Currying

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Recursion Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Isomorphic Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Function Composition

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Function Naming

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Generics

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Callable Types

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Comprehensions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Integrated Optionals

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Optional Chaining

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## A Native Bind Operator

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Enumerated Types

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

# When to Go Functional

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Avoid Global State

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Make Pure Functions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Isolate IO

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## List Application

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Embrace Value Objects

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Embrace Functions as Values

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Make Single-Method Callable Services

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Think in Terms of Pipelines and Data Flow

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Use Maybe and Either for Error Handling

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Don’t Overdo It

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

# Further Resources

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Books

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Videos

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.

## Papers

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/thinking-functionally-in-php.