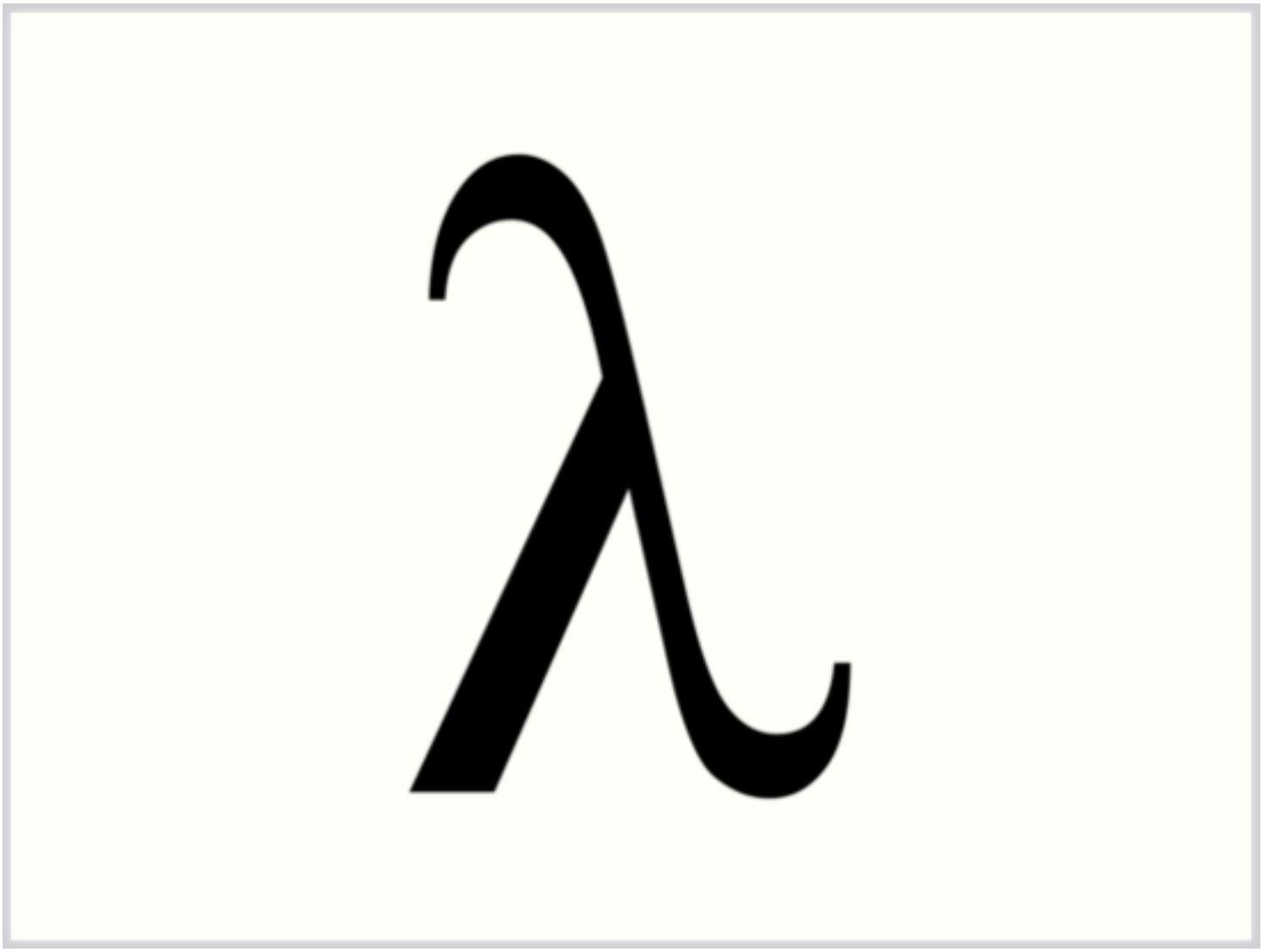
15 NOVEMBER 2018 / #FUNCTIONAL PROGRAMMING

An Introduction to the basic principles of Functional Programming



by TK

After a long time learning and working with object-oriented programming, I took a step back to think about system complexity.

> "`Complexity is anything that makes software hard to understand or to modify.`" — John Outerhout

Doing some research, I found functional programming concepts like immutability and pure function. Those concepts are big advantages to build side-effect-free functions, so it is easier to maintain systems — with some other benefits.

In this post, I will tell you more about functional programming, and some important concepts, with a lot of code examples.

This article uses Clojure as a programming language example to explain Functional Programming. If you are not comfortable with a LISP-type-of-language, I also published the same post in JavaScript. Take a look: **Functional Programming Principles in Javascript**

What is functional programming?

> **Functional programming** is a programming paradigm — a style of building the structure and elements of computer programs — that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data — Wikipedia

Pure functions

"water drop" by Mohan Murugesan on Unsplash

The first fundamental concept we learn when we want to understand functional programming is **pure functions**. But what does that really mean? What makes a function pure?

So how do we know if a function is `pure` or not? Here is a very strict definition of purity:

- It returns the same result if given the same arguments (it is also referred as `deterministic`)
- It does not cause any observable side effects

It returns the same result if given the same arguments

Imagine we want to implement a function that calculates the area of a circle. An impure function would receive `radius` as the parameter, and then calculate `radius * radius * PI`. In

becomes `(* radius radius PI)`:
Why is this an impure function? Simply because it uses a global object that was not passed as a parameter to the function.

Now imagine some mathematicians argue that the `PI` value is actually `42` and change the value of the global object.

Our impure function will now result in `10 * 10 * 42` = `4200`. For the same parameter (`radius = 10`), we have a different result. Let's fix it!

TA-DA ?! Now we'll always pass the`PI` value as a parameter to the function. So now we are just accessing parameters passed to the function. No e`xternal object.`

- For the parameters `radius = 10` & `PI = 3.14`, we will always have the same the result: `314.0`
- For the parameters `radius = 10` & `PI = 42`, we will always have the same the result: `4200`

Reading Files

If our function reads external files, it's not a pure function — the file's contents can change.

Random number generation

Any function that relies on a random number generator cannot be pure.

It does not cause any observable side effects

Examples of observable side effects include modifying a global object or a parameter passed by reference.

Now we want to implement a function to receive an integer value and return the value increased by 1.

We have the `counter` value. Our impure function receives that value and re-assigns the counter with the value increased by 1.

Observation: mutability is discouraged in functional programming.

We are modifying the global object. But how would we make it `pure`? Just return the value increased by 1. Simple as that.

See that our pure function `increase-counter` returns 2, but the `counter` value is still the same. The function returns the incremented value without altering the value of the variable.

If we follow these two simple rules, it gets easier to understand our programs. Now every function is isolated and unable to impact other parts of our system.

Pure functions are stable, consistent, and predictable. Given the same parameters, pure functions will always return the same result. We don't need to think of situations when the

happen.

Pure functions benefits

The code's definitely easier to test. We don't need to mock anything. So we can unit test pure functions with different contexts:

- Given a parameter `A` → expect the function to return value `B`
- Given a parameter `C` → expect the function to return value `D`

A simple example would be a function to receive a collection of numbers and expect it to increment each element of this collection.

We receive the `numbers` collection, use `map` with the `inc` function to increment each number, and return a new list of incremented numbers.

For the `input` `[1 2 3 4 5]`, the expected `output` would be `[2 3 4 5 6]`.

Immutability

> *Unchanging over time or unable to be changed.*

"Change neon light signage" by Ross Findon on Unsplash

When data is immutable, its **state cannot change after it's created.** If you want to change an immutable object, you can't. Instead, **you create a new object with the new value.**

In Javascript we commonly use the `for` loop. This next `for` statement has some mutable variables.

For each iteration, we are changing the `i` and the `sumOfValue` **state**. But how do we handle mutability in iteration? Recursion! Back to Clojure!

So here we have the `sum` function that receives a vector of numerical values. The `recur` jumps back into the `loop` until we get the vector empty (our recursion `base case`). For each "iteration" we will add the value to the `total` accumulator.

With recursion, we keep our **variables** immutable.

Observation: Yes! We can use `reduce` to implement this function. We will see this in the `Higher Order Functions` topic.

Imagine we have a string, and we want to transform this string into a `url slug`.

In OOP in Ruby, we would create a class, let's say, `UrlSlugify`. And this class will have a `slugify!` method to transform the string input into a `url slug`.

Beautiful! It's implemented! Here we have imperative programming saying exactly what we want to do in each `slugify` process — first lower case, then remove useless white spaces and, finally, replace remaining white spaces with hyphens.

But we are mutating the input state in this process.

We can handle this mutation by doing function composition, or function chaining. In other words, the result of a function will be used as an input for the next function, without modifying the original input string.

Here we have:

- `trim`: removes whitespace from both ends of a string
- `lower-case`: converts the string to all lower-case
- `replace`: replaces all instances of match with replacement in a given string

We combine all three functions and we can `"slugify"` our string.

Speaking of **combining functions**, we can use the `comp` function to compose all three functions. Let's take a look:

Referential transparency

"person holding eyeglasses" by Josh Calabrese on Unsplash

Let's implement a `square function`:

This (pure) function will always have the same output, given the same input.

Passing "2" as a parameter of the `square function` will always returns 4. So now we can replace the `(square 2)` with 4. That's it! Our function is `referentially transparent`.

Basically, if a function consistently yields the same result for the same input, it is referentially transparent.

pure functions + immutable data = referential transparency

With this concept, a cool thing we can do is to memoize the function. Imagine we have this function:

The `(+ 5 8)` equals `13`. This function will always result in `13`. So we can do this:

And this expression will always result in `16`. We can replace the entire expression with a numerical constant and memoize it.

Functions as first-class entities

"first-class" by Andrew Neel on Unsplash

also treated as values **and** used as data.

In Clojure it's common to use `defn` to define functions, but this is just syntactic sugar for `(def foo (fn ...))`. `fn` returns the function itself. `defn` returns a `var` which points to a function object.

Functions as first-class entities can:

- refer to it from constants and variables
- pass it as a parameter to other functions
- return it as result from other functions

The idea is to treat functions as values and pass functions like data. This way we can combine different functions to create new functions with new behavior.

Imagine we have a function that sums two values and then doubles the value. Something like this:

Now a function that subtracts values and the returns the double:

These functions have similar logic, but the difference is the operators functions. If we can treat functions as values and pass these as arguments, we can build a function that receives the operator function and use it inside our function. Let's build it!

Done! Now we have an `f` argument, and use it to process `a`

`double-operator` function and create a new behavior.

Higher-order functions

When we talk about higher-order functions, we mean a function that either:

- takes one or more functions as arguments, or
- returns a function as its result

The `double-operator` function we implemented above is a higher-order function because it takes an operator function as an argument and uses it.

You've probably already heard about `filter`, `map`, and `reduce`. Let's take a look at these.

Filter

Given a collection, we want to filter by an attribute. The filter function expects a `true` or `false` value to determine if the element **should or should not** be included in the result collection. Basically, if the callback expression is `true`, the filter function will include the element in the result collection. Otherwise, it will not.

A simple example is when we have a collection of integers and we want only the even numbers.

Imperative approach

- create an empty vector `evenNumbers`
- iterate over the `numbers` vector
- push the even numbers to the `evenNumbers` vector

We can use the `filter` higher order function to receive the `even?` function, and return a list of even numbers:

One interesting problem I solved on Hacker Rank FP Path was the **Filter Array problem**. The problem idea is to filter a given array of integers and output only those values that are less than a specified value `X`.

An imperative Javascript solution to this problem is something like:

We say exactly what our function needs to do — iterate over the collection, compare the collection current item with `x`, and push this element to the `resultArray` if it pass the condition.

Declarative approach

But we want a more declarative way to solve this problem, and using the `filter` higher order function as well.

A declarative Clojure solution would be something like this:

This syntax seems a bit strange in the first place, but is easy to understand.

`#(> x %)` is just a anonymous function that receives x and compares it with each element in the collection. % represents the parameter of the anonymous function — in this case the current element inside the filter.

We can also do this with maps. Imagine we have a map of people with their `name` and `age`. And we want to filter only people over a specified value of age, in this example people who are more than 21 years old.

Summary of code:

- we have a list of people (with `name` and `age`).
- we have the anonymous function `#(< 21 (:age %))`. Remember that the % represents the current element from the collection? Well, the element of the collection is a people map. If we `do (:age {:name "TK" :age 26})`, it returns the age value, 26 in this case.
- we filter all people based on this anonymous function.

Map

The idea of map is to transform a collection.

> The *`map`* method transforms a collection by applying a function to all of its elements and building a new collection from the returned values.

Let's get the same `people` collection above. We don't want to

something like `TK is 26 years old`. So the final string might be `:name is :age years old` where `:name` and `:age` are attributes from each element in the `people` collection.

In a imperative Javascript way, it would be:

In a declarative Clojure way, it would be:

The whole idea is to transform a given collection into a new collection.

Another interesting Hacker Rank problem was the **update list problem**. We just want to update the values of a given collection with their absolute values.

For example, the input `[1 2 3 -4 5]` needs the output to be `[1 2 3 4 5]`. The absolute value of `-4` is `4`.

A simple solution would be an in-place update for each collection value.

We use the `Math.abs` function to transform the value into its absolute value, and do the in-place update.

This is **not** a functional way to implement this solution.

First, we learned about immutability. We know how immutability is important to make our functions more consistent and predictable. The idea is to build a new collection with all absolute values.

My first idea was to build a `to-absolute` function to handle only one value.

If it is negative, we want to transform it in a positive value (the absolute value). Otherwise, we don't need to transform it.

Now that we know how to do `absolute` for one value, we can use this function to pass as an argument to the `map` function. Do you remember that a `higher order function` can receive a function as an argument and use it? Yes, map can do it!

Wow. So beautiful! ?

Reduce

The idea of reduce is to receive a function and a collection, and return a value created by combining the items.

A common example people talk about is to get the total amount of an order. Imagine you were at a shopping website. You've added `Product 1`, `Product 2`, `Product 3`, and `Product 4` to your shopping cart (order). Now we want to calculate the total amount of the shopping cart.

In imperative way, we would iterate the order list and sum each product amount to the total amount.

Using `reduce`, we can build a function to handle the `amount su m` and pass it as an argument to the `reduce` function.

Here we have `shopping-cart`, the function `sum-amount` that receives the current `total-amount`, and the `current-product` object to `sum` them.

The `get-total-amount` function is used to `reduce` the `shoppin g-cart` by using the `sum-amount` and starting from `0`.

Another way to get the total amount is to compose `map` and `re duce`. What do I mean by that? We can use `map` to transform the `shopping-cart` into a collection of `amount` values, and then just use the `reduce` function with `+` function.

The `get-amount` receives the product object and returns only the `amount` value. So what we have here is `[10 30 20 60]`. And then the `reduce` combines all items by adding up. Beautiful!

We took a look at how each higher-order function works. I want to show you an example of how we can compose all three functions in a simple example.

Talking about `shopping cart`, imagine we have this list of products in our order:

We want the total amount of all books in our shopping cart. Simple as that. The algorithm?

- **filter** by book type
- transform the shopping cart into a collection of amount using **map**
- combine all items by adding them up with **reduce**

Done! ?

Resources

I've organised some resources I read and studied. I'm sharing the ones that I found really interesting. For more resources, visit my **Functional Programming Github repository**.

- Ruby specific resources
- Javascript specific resources
- Clojure specific resources

Intros

- Learning FP in JS
- Intro do FP with Python
- Overview of FP
- A quick intro to functional JS
- What is FP?
- Functional Programming Jargon

Pure functions

- What is a pure function?
- Pure Functional Programming 1
- Pure Functional Programming 2

Immutable data

- Immutable DS for functional programming
- Why shared mutable state is the root of all evil
- Structural Sharing in Clojure: Part 1
- Structural Sharing in Clojure: Part 2
- Structural Sharing in Clojure: Part 3
- Structural Sharing in Clojure: Final part

Higher-order functions

- Eloquent JS: Higher Order Functions
- Fun fun function Filter
- Fun fun function Map
- Fun fun function Basic Reduce
- Fun fun function Advanced Reduce
- Clojure Higher Order Functions
- Purely Function Filter
- Purely Functional Map
- Purely Functional Reduce

Declarative Programming

- Declarative Programming vs Imperative

That's it!

you learned a lot here! This was my attempt to share what I'm learning.

Here is the repository with all codes from this article.

Come learn with me. I'm sharing resources and my code in this **Learning Functional Programming repository**.

I hope you saw something useful to you here. And see you next time! :)

My Twitter & Github. ☺

TK.

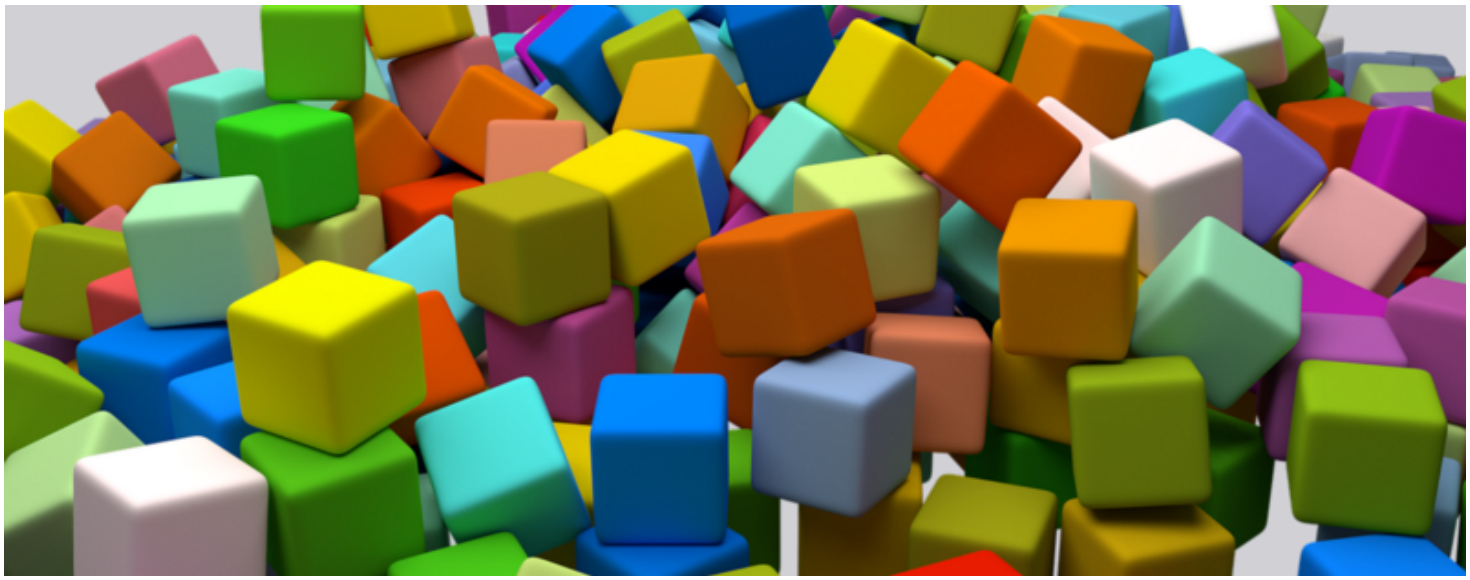
Countinue reading about

Functional Programming

Learning functional programming made me a 10x better developer

Yet Another 10 Utility Functions Made with Reduce

See all 97 posts →



#PROGRAMMING

Let's get classy: how to create modules and classes with Python

A YEAR AGO

#BLOCKCHAIN

How to build a simple actor-based blockchain

LUCA FLORIO A YEAR AGO

freeCodeCamp is a donor-supported tax-exempt 501(c)(3) nonprofit organization (United States Federal Tax Identification Number: 82-0779546)

Our mission: to help people learn to code for free. We accomplish this by creating thousands of videos, articles, and interactive coding lessons - all freely available to the public. We also have thousands of freeCodeCamp study groups around the world.

Donations to freeCodeCamp go toward our education initiatives, and help pay for servers, services, and staff.

You can make a tax-deductible donation here.

Our Nonprofit

About

Alumni Network

Open Source

Shop

Support

Sponsors

Academic Honesty

Code of Conduct

Privacy Policy

Terms of Service

Copyright Policy

Trending Guides

2019 Web Developer Roadmap

Python Tutorial

CSS Flexbox Guide

JavaScript Tutorial

Python Example

HTML Tutorial

Linux Command Line Guide

JavaScript Example

Git Tutorial

React Tutorial

Java Tutorial

Linux Tutorial

CSS Tutorial

jQuery Example

SQL Tutorial

CSS Example

React Example

Angular Tutorial

Bootstrap Example

How to Set Up SSH Keys

WordPress Tutorial

PHP Example