Functional

Programming with JS

ARIHARASUDHAN

This Book

This book is written (typed?) by Ari, who hails from The South and has keen interests in Computer Science, Biology, and Tamil Literature. Occasionally, he updates his website where you can reach out to him.
https://arihara-sudhan.github.io

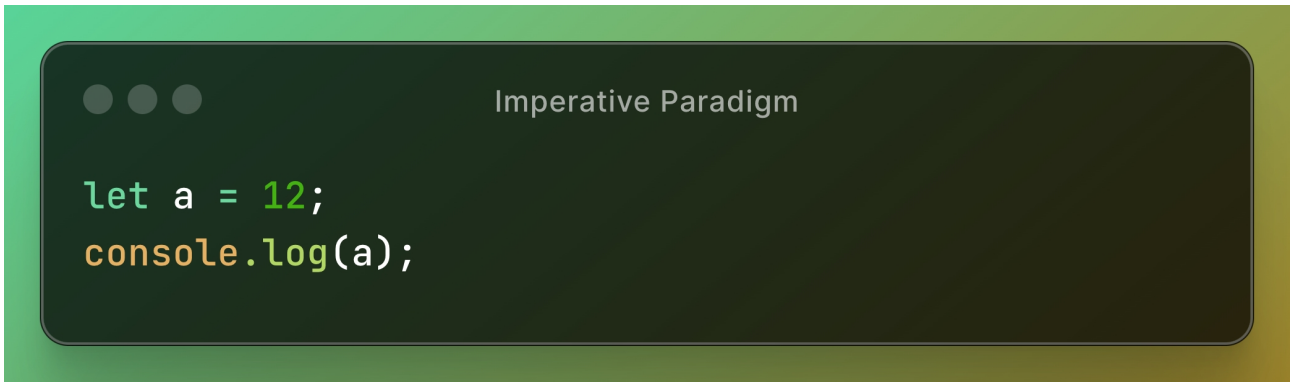
Functional Programming

Functional Programming is a programming paradigm where functions are TIGERS (I wanted to say KINGS; Tigers came to my mind... So, I concluded with Tigers, as Tiger is better suited to be king of the jungle than the so called lion, the hairy faced cat.)

So, where to start? How to code in functional programming paradigm? Let's start right now!

Functional Paradigm (FP)

When someone say the word “Functional Programming”, we may guess something like, “doing with functions”. For your astonishment, that is the core of this paradigm. Yes! Expressing everything in our program in terms of functions is the functional programming paradigm. There is another paradigm called as Imperative Programming in which, we specify what to do imperatively.



Here, in this snippet, we say, do this (let a = 12) and then do this (console.log(a)). This is imperative!

This is an example of imperative programming paradigm. We don't use functions to achieve the task here. To make it functional, we''ll write like,

Functional Paradigm

```
function log(v){
  console.log(v);
}

log(12);
```

This is a functional way! Isn't it? Solely using functions doesn't means we are using functional programming paradigm. Our functions should avoid causing side effects.

Pure Functions

A function may cause side effects. If a function needs to access a globally defined variable to perform a particular action, it means that the function is not pure. In this case, the reading of the global variable is a side effect caused by the function. A pure function solely depends on its input and never does any side effects. It just accepts inputs, uses that and only that and finally returns the output. In the following code, the function causes a side effect by accessing a globally defined variable.

Impure

```
var name = "Ariharasudhan";

function beImPure(){
  console.log(`Hello ${name}`);
}
```

To make it pure,

Pure

```
function beImPure(name){
  return `Hello ${name}`;
}
```

The first snippet has an impure function. It is impure as it depends on a globally defined variable. It doesn’t even accept any inputs. The second snippet has a pure function. It accepts an input (solely depends on that input alone) and returns the output. If you have watched the following, you can realize, that baby is a pure function.

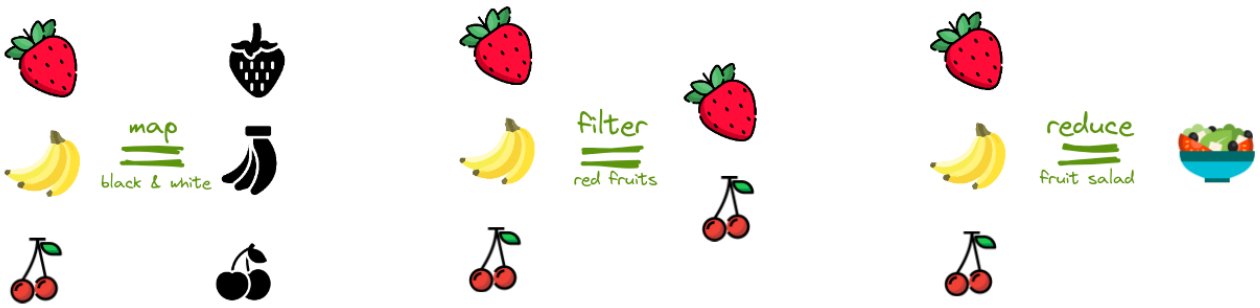
Higher Order Functions

In JS, as in Python, as in Clojure, as in Haskell, as in Java (since 8 with non-verbose Lambdas) and so on, a function can be passed to another function and also can be returned from another function or A function can accept and also return another function.

This is called Higher Order Function. Wait a minute! Which is called higher order function? The one which accepts function(s)/returns function(s)? Or, The one which is passed to another function? The function that accepts or returns other function(s) is the so called, “Higher Order Function”. The function that is passed to another function is... Guess what... The so called, “Call-backs”. This is a key concept here in FP!

NO Iterations

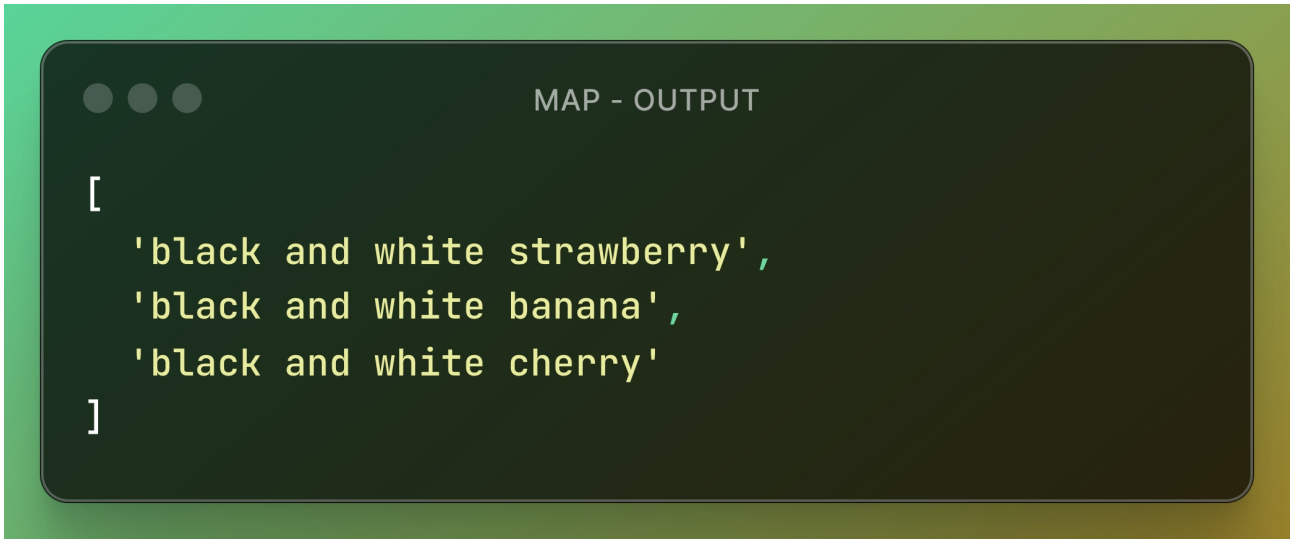
In Functional Paradigm, it is strictly followed custom to avoid iterations. Instead, we might use the higher order functions such as Map, Filter, Reduce and so on. Most of'em takes a list and a custom function to apply on that list.



Map is a higher order function. It accepts a callback and applies that on the list.

MAP

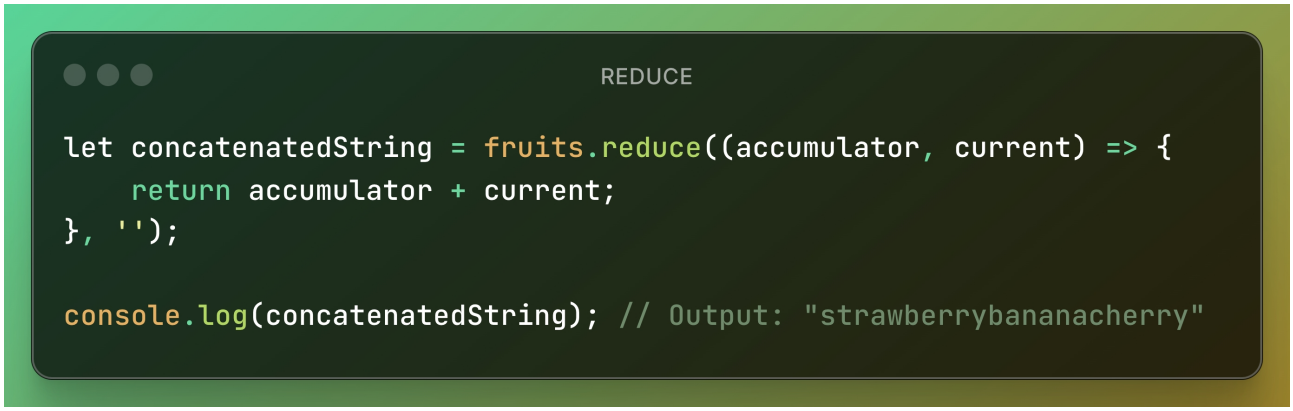
```
let fruits = ["strawberry", "banana", "cherry"];
let bwfruits = fruits.map(fruit=>{
    return "black and white "+fruit
})

console.log(bwfruits);
```



Filter is another higher order function that returns a filtered list. If I need only the fruits whose names end in “rry”, I can write a filter function as shown below.

Now comes the Reduce, a higher order function that accumulates the data in the list.

If I want to concatenate all fruit names into a single name, I can use a reduce function accordingly.



If we want to perform these all without using these higher order functions, imagine the situation!! Writing a for loop to take each value and checking them... putting them in a list... blah blah blah... This is the so called, “Imperative Style”. In functional paradigm, it is so declarative with these Higher Order Functions.

NO Mutations

Mutation is non functional. Sometimes, If we change a data, it may rise lot of problems in our code. It may affect the integrity of the program if the same data has to be changed in lot of sections and left unchanged. Functional Paradigm avoids this by introducing immutability.

OP: ['Ari', 'Haran', 'Aravindhan']

To make it immutable, we can use the higher-order function Map in the following way.

IMMUTABILITY

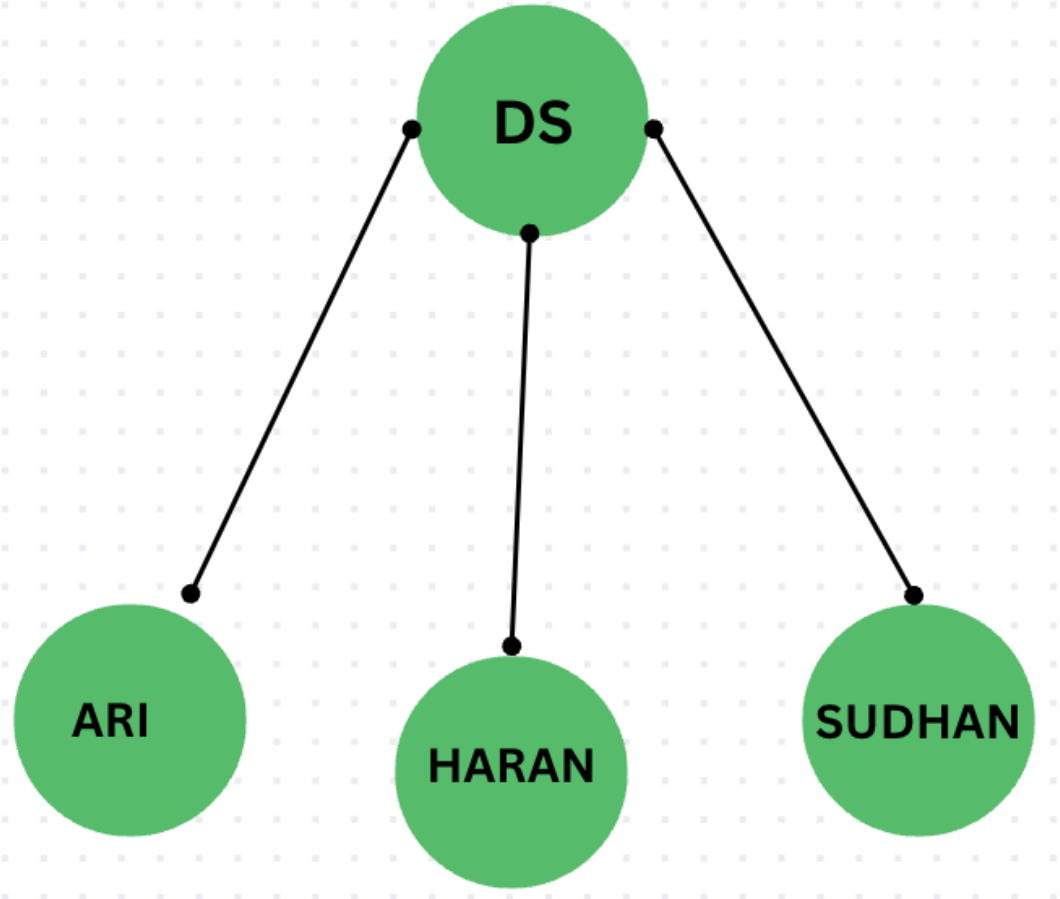
```
let names = ["Ari", "Haran", "Sudhan"];
let changed = names.map((name)=>{
    return name === "Sudhan"? "Aravindhan": name;
});
console.log(changed);
```

OP: ['Ari', 'Haran', 'Aravindhan']

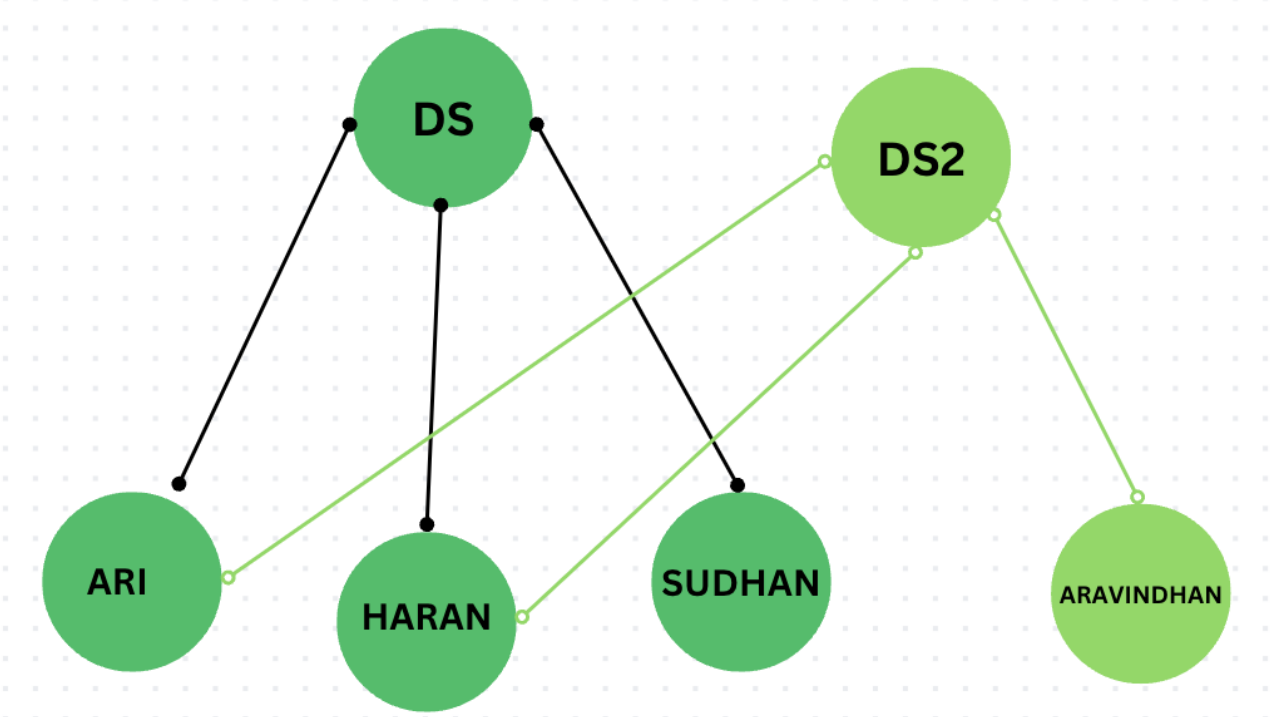
Now, the actual list is not changed. We have a copy of the list where only the intended data is changed. But, is it okay to copy over these data? Will it be efficient for large problems? Contemplate! Just to change one data in the list, should we create a copy of the existing list? Hello Nah!

Persistent Datastructures

As a way of getting around the problem discussed above, functional style uses persistent datastructures. It is to reduce the space and time used in the conventional way of achieving immutability. How can we do this? We can represent the list of data as a tree as shown below.



All we need to do is to create a new node (root) that points to the existing data and a changed data.



It is so evident! We didn't create a changed copy of existing elements. Instead, we simply linked the new data with the existing. DS2 is the efficiently mutated collection. But, the actual data is not changed. We achieve immutability here!

There are lot of libraries that provides such persistent datastructures. Some of them are, Ramda, Lodash, Underscore, Immutable.js, Mori and so on.

Essence

> Use Functions
> Pure Functions I meant
> Higher Order Functions are so usual
> Don’t Iterate
> Be declarative than imperative
> Immutability
> Persistent Datatstructures

MERCI