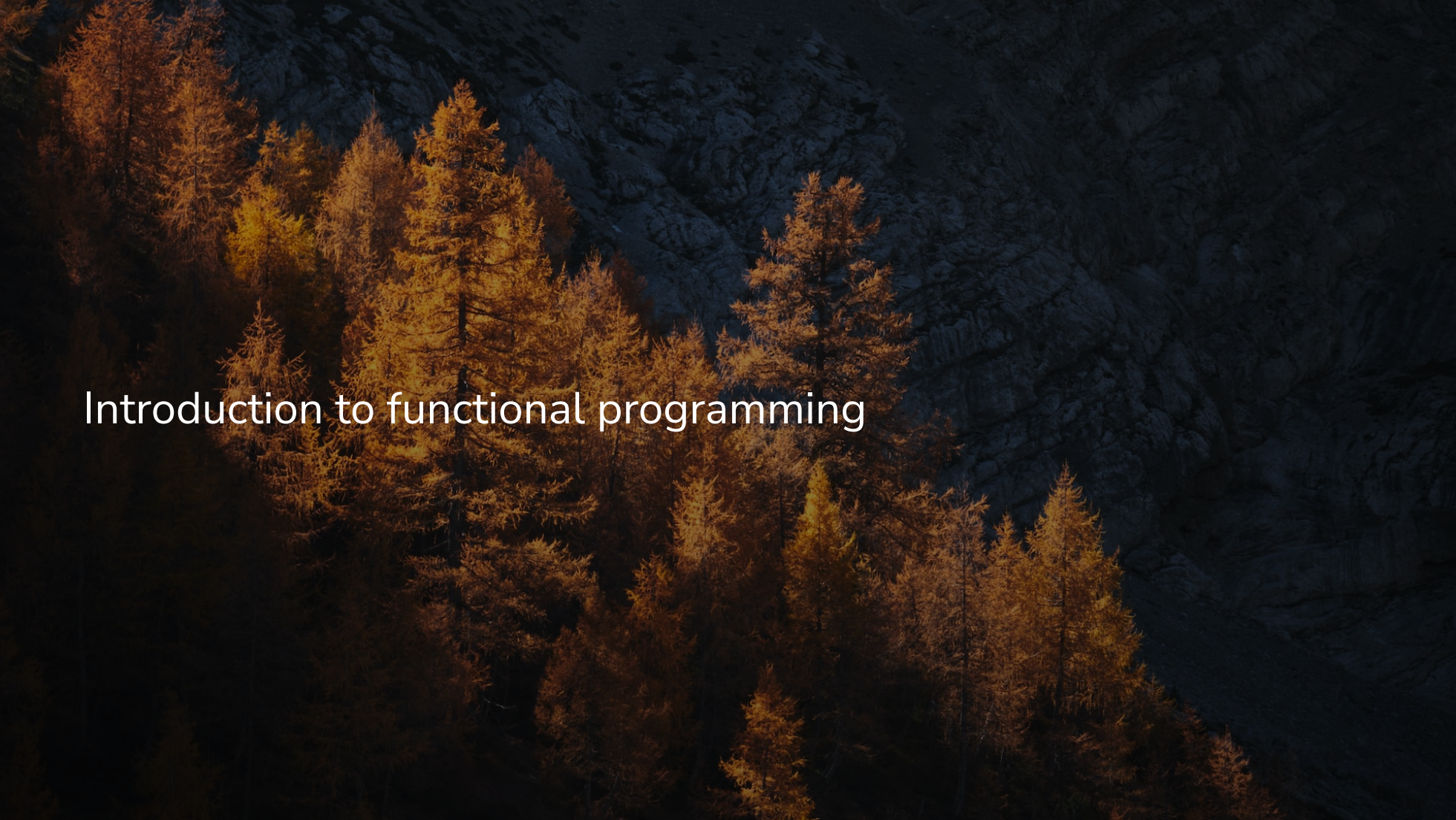

Introduction to functional programming

Table of contents

Motivation

A language that doesn’t affect the way you think about programming, is not worth knowing

Alan Perlis

Functional programming is NOT

1. something new or recent
2. a different language
3. solution for all our problems
4. very complex or difficult
5. accessible only for people with good understanding of math
6. only applicable for math-related problems
7. incompatible with other paradigms, like imperative or object oriented programming

Why should we learn that?

1. completely different way of thinking about problems
2. very generic and portable (you don't need to understand language specifics, like `this`, `self` keywords, prototypal inheritance, etc.)
3. easier to parallelize
4. easier to test
5. easier to verify
6. it's becoming more mainstream (i.e. pattern matching in Python 3.10, promises in ES6)
7. expressive, declarative (and hence more readable)

Quick Sort Example - Imperative (JavaScript)

```
function partition(items, left, right) {
    let p = items[0];
    let i = left
    let j = right;
    while (i <= j) {
        while (items[i] < pivot) { i++; }
        while (items[j] > pivot) { j--; }
        if (i <= j) {
            [ items[i], items[j] ] = [ items[j], items[i] ];
            i++;
            j--;
        }
    }
    return i;
}
function quicksort(items, left, right) {
    let index;
    if (items.length > 1) {
        index = partition(items, left, right);
        if (left < index - 1) { quicksort(items, left, index - 1); }
        if (index < right) { quicksort(items, index, right); }
    }
    return items;
}
```

Quick Sort Example - Functional (JavaScript & Haskell)

JavaScript:

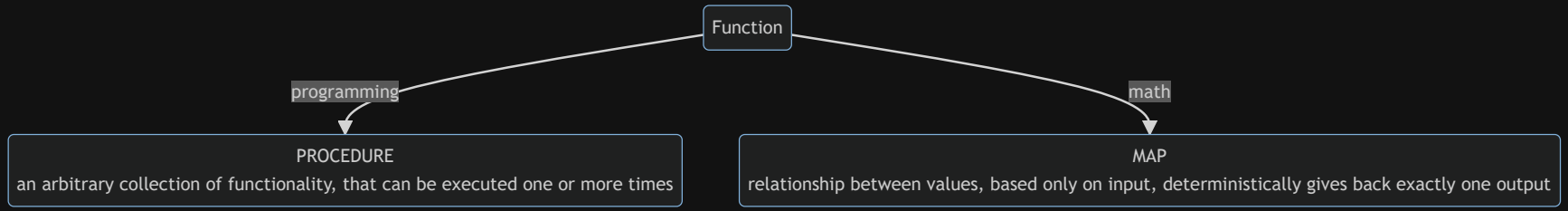
```javascript
function quicksort(array) {
  if (!array.length) {
    return [];
  };
  const [pivot, ...rest] = array;
  const lesser = rest.filter(i => i < pivot);
  const greater = rest.filter(i => i >= pivot);
  return [...quicksort(lesser), pivot, ...quicksort(greater)];
}
```

Haskell:

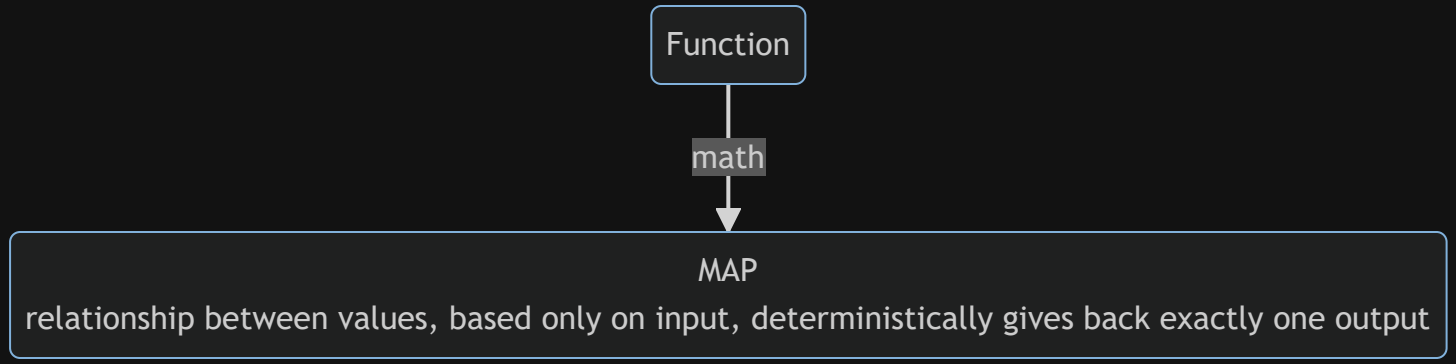
```haskell
quicksort []     = []
quicksort (pivot:rest) = (quicksort lesser) ++ [p] ++ (quicksort greater)
    where
        lesser  = filter (< pivot) rest
        greater = filter (>= pivot) rest
```

Basic concepts

Functional programming - programming with functions



Functional programming - programming with functions



Usually we call these "pure functions".

Implications of this definition:

- pure (no observable side effects)
- easy to parallelize (no shared state)
- idempotent & cacheable (you can memorize the previous outputs)
- testable (forget about mocks or assertions of shared state)
- reliable, easier to debug (referential transparency)
- reusable, composable, portable

Side Effects = "he who must not be named"

Let’s play - pure function or not a pure function?

JavaScript

```javascript
function increment(a) {
  return a + 1;
}
```

```javascript
function birthday(user) {
  user.age += 1
  return user;
}
```

```javascript
function shout(sentence) {
  return sentence.toUpperCase() + '!';
}
```

```javascript
function parseQueryString(query) {
    return query
      .substring(1)
      .split('&')
      .map(x => x.split('='))
}
```

Python

```python
def increment(a):
  return a + 1
```

```python
def birthday(user):
  user.age += 1
  return user
```

```python
def shout(sentence):
  return sentence.upper() + '!'
```

```python
def parseQueryString(query):
  return [x.split('=') for x in query[1:].split('&')]
```

JavaScript

```javascript
function signUp(input) {
  const user = dbProvider.save(input);
  emailProvider.sendWelcomeEmail(user);
}
```

```javascript
function getLastElement(arr) {
  return arr.pop();
}
```

```javascript
function getLastElement2(arr) {
  return arr.at(-1);
}
```

```javascript
function add(a, b) {
  console.log(`Adding ${a} + ${b}`);
  return a + b;
}
```

Python

```python
def signUp(input):
  user = dbProvider.save(input)
  emailProvider.sendWelcomeEmail(user)
```

```python
def getLastElement(arr):
  return arr.pop()
```

```python
def getLastElement2(arr):
  return arr[-1]
```

```python
def add(a, b):
  print(f'Adding {a} + {b}')
  return a + b
```

Why do we care so much about purity?

Higher-order functions

First Class Functions

feature of programming language, thanks to which functions in that language are treated like any other variable.

Higher order function

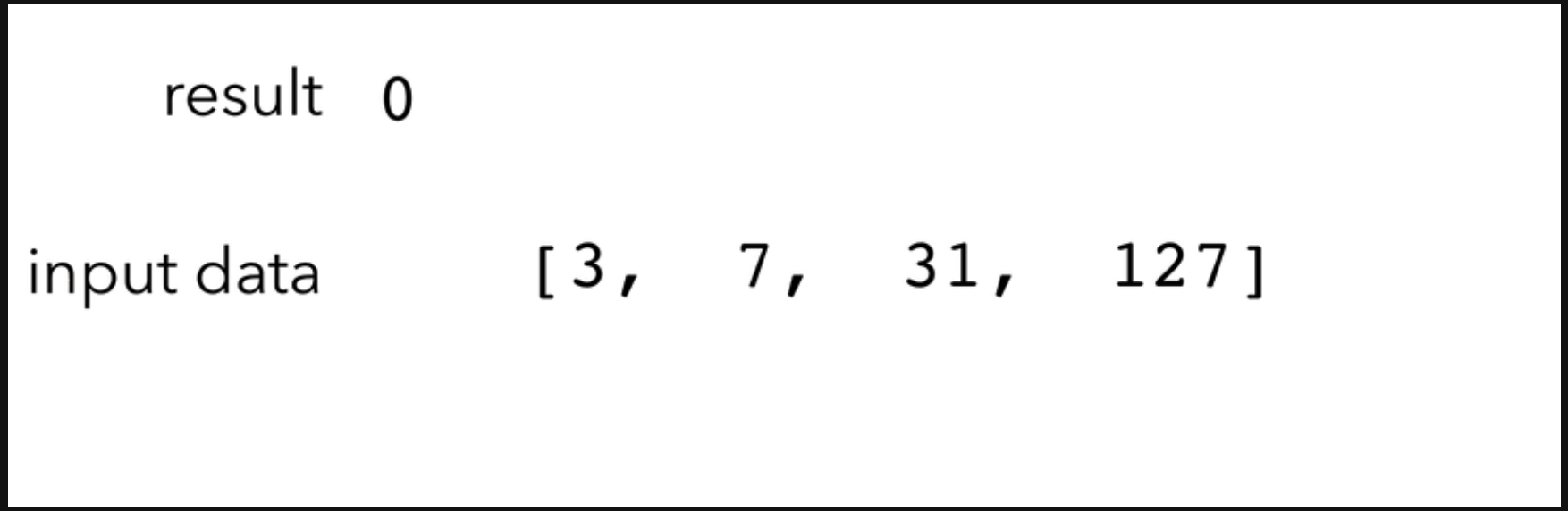
function that does at least one of the following: accepts another function as parameter, or returns a function as its result

Reduce function:

```
[3, 7, 31, 127].reduce((acc, item) => acc + item, 0);
```

```
reduce(lambda acc, item: acc + item, [3, 7, 31, 127], 0)
```

How it works:



Point free style

coding style, where caller does not pass arguments explicitly to a function

```javascript
function isReleasedIn90s(song) {
  return 1990 <= song.released && song.released <= 1999;
}

const songs = [
  { name: 'Space Oddity', released: 1972 },
  { name: 'Karma Police', released: 1997 },
  { name: 'Pictures Of You', released: 1993 },
  { name: 'Wonderwall', released: 1995 },
  { name: 'Let It Happen', released: 2015 },
  { name: 'Where Is My Mind', released: 1988 },
];

// Normal
filter((song) => isReleasedIn90s(song), songs);

// Point free style
filter(isReleasedIn90s, songs);
```

```python
def isReleasedIn90s(song):
  return 1990 <= song['released'] and song['released'] <= 1999


songs = [
  { 'name': 'Space Oddity', 'released': 1972 },
  { 'name': 'Karma Police', 'released': 1997 },
  { 'name': 'Pictures Of You', 'released': 1993 },
  { 'name': 'Wonderwall', 'released': 1995 },
  { 'name': 'Let It Happen', 'released': 2015 },
  { 'name': 'Where Is My Mind', 'released': 1988 },
]

# Normal
filter(lambda song: isReleasedIn90s(song), songs)

# Point free style
filter(isReleasedIn90s, songs)
```

Currying

Curried Function

A function that will return a new function until it receives all it's arguments.

```javascript
function add(x, y) {
  return x + y;
}

const a = add(1, 2); // 3
```

```python
def add(x, y):
  return x + y


a = add(1, 2) # 3
```

```javascript
function addCurried(x) {
  return function(y) {
    return x + y;
  }
}

const b = addCurried(1)(2); // 3
```

```python
def addCurried(x):
  def addCurriedInner(y):
    return x + y
  return addCurriedInner


b = addCurried(1)(2) # 3
```

Why the hell would we do that?

It allows us to perform partial application, making it possible to save some of the arguments to the function.

Partial application

Giving a function fewer arguments than it expects.

```javascript
function add(x, y) {
  return x + y;
}

function increment(x) {
  return add(1, x);
}

function decrement(x) {
  return add(-1, x);
} // we need to redeclare functions...
```

```python
def add(x, y):
  return x + y


def increment(x):
  return add(1, x)


def decrement(x):
  return add(-1, x)
# we need to redeclare functions...
```

```javascript
function addCurried(x) {
  return function(y) {
    return x + y;
  }
}
const incrementCurried = addCurried(1); // one-liner!
const decrementCurried = addCurried(-1);
```

```python
def addCurried(x):
  def addCurriedInner(y):
    return x + y
  return addCurriedInner

incrementCurried = addCurried(1) # one-liner!
decrementCurried = addCurried(-1)
```

curry

a higher-order function, that transforms normal function to a curried function

```javascript
const curriedAdd = curry((x, y, z) => x + y + z);

curriedAdd(1, 2, 3); // 6
curriedAdd(1, 2)(3); // 6
curriedAdd(1)(2, 3); // 6
curriedAdd(1)(2)(3); // 6
```

```python
curriedAdd = curry(lambda x, y, z: x + y + z)

curriedAdd(1, 2, 3) # 6
curriedAdd(1, 2)(3) # 6
curriedAdd(1)(2, 3) # 6
curriedAdd(1)(2)(3) # 6
```

Order of function parameters matters!

First provide these, that might be worth saving for later use.

```javascript
// (probably) wrong
function filter(arr, predicate) { /* ... */ }

function inRange(val, minimum, maximum) { /* ... */ }


// (probably) better
function filter(predicate, arr) { /* ... */ }

const _filter = curry(filter);

function inRange(min, max, val) { /* ... */ }

const _inRange = curry(inRange);

const betweenZeroAndHundred = _filter(_inRange(0, 100));
betweenZeroAndHundred([1, 200, 3, -5, 40]); // -> [1, 3, 40]
betweenZeroAndHundred([100, 200, 300]); // -> []
```

```python
# (probably) wrong
def filter(arr, predicate):
  pass
def inRange(val, minimum, maximum):
  pass

# (probably) better
def filter(predicate, arr):
  pass
_filter = curry(filter)

def inRange(min, max, val):
  pass
_inRange = curry(inRange)

betweenZeroAndHundred = _filter(_inRange(0, 100))
betweenZeroAndHundred([1, 200, 3, , -5, 40]) # -> [1, 3, 40]
betweenZeroAndHundred([100, 200, 300]) # -> []
```

Composition

Function composition

operation, that takes two functions: `f` and `g`, and produces a function, that when called with argument `x` is equivalent to calling `f(g(x))`

```
const toUpperCase = x => x.toUpperCase();
const exclaim = x => `${x}!`;
const shout = compose(exclaim, toUpperCase);

shout('send in the clowns'); // -> "SEND IN THE CLOWNS!"
```

```
toUpperCase = lambda x: x.upper()
exclaim = lambda x: f'{x}!'
shout = compose(exclaim, toUpperCase)

shout('send in the clowns') # -> "SEND IN THE CLOWNS!"
```

Unfortunately, by definition `compose` works only on two functions...

BUT!

Composition is associative!

`compose(compose(f, g), h)` is the same as `compose(f, compose(g, h))`

it doesn't matter how we group the compositions

So we can define:

`compose(f, g, h)` as `compose(f, compose(g, h))`

and generalize it to any number of functions!

Great!

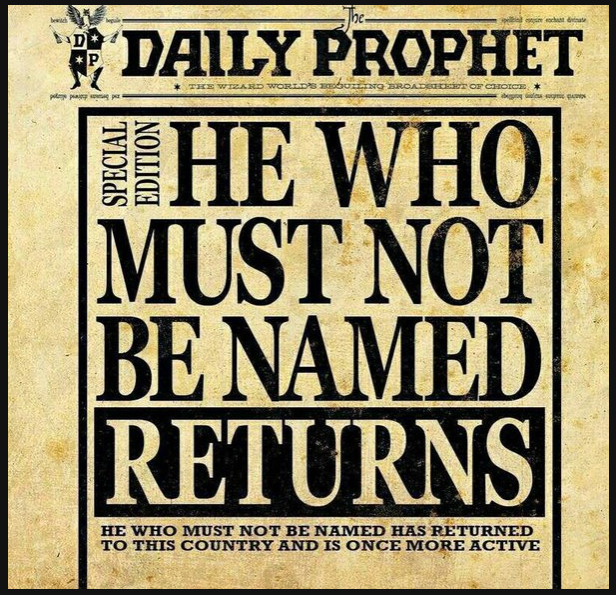
Now we have a very powerful tool, that gives us a very convenient way to "combine" functions!

Functors and Monads

What we learned so far is really powerful...

really, really, really powerful...

but we forgot about one small little thing... can you guess what?



Applications are all about side effects...

We all know arrays. We can map them:

```
map((name) => name.length, ['Harry', 'Hermiona', 'Ron']);
```

```
map((name) => len(name), ['Harry', 'Hermiona', 'Ron'])
```

Array is some kind of *container* (that holds multiple values).

`map` is a higher-order function, that allows us to perform some other function on that *container*

Container decides how this function is applied.

Maybe we can generalize...?

Functor is a box

type that implements `map` method (sometimes called `fmap` or `<$>`) and obeys two laws. For functor `F`:

1. Mapping with identity function does not modify the value

```javascript
const identity = (x) => x;
ArrayF.of(1, 2, 3).map(identity);
// should contain the same value as:
ArrayF.of(1, 2, 3);
// -> [1, 2, 3]
```

```python
identity = lambda x: x
ArrayF.of(1, 2, 3).map(identity)
# should contain the same value as:
ArrayF.of(1, 2, 3)
# -> [1, 2, 3]
```

2. Mapping with `f` and then `g` is the same as mapping with composition `compose(f, g)`

```javascript
const double = (x) => x * 2;
const inc = (x) => x + 1;
const double_and_inc = compose(inc, double);
ArrayF.of(1, 2, 3).map(double).map(inc);
// should contain the same value as:
ArrayF.of(1, 2, 3).map(double_and_inc);
// -> [3, 5, 7]
```

```python
double = lambda x: x * 2
inc = lambda x: x + 1
double_and_inc = compose(inc, double)
ArrayF.of(1, 2, 3).map(double).map(inc)
# should contain the same value as F
ArrayF.of(1, 2, 3).map(double_and_inc)
# -> [3, 5, 7]
```

Monad is an upgraded box

it is a functor that also implements `chain` method (sometimes called `flatMap` or `>>=`)

- we use `map`, when we want to apply a function, which does not use the fact, that we are working with monad and returns unwrapped value: `(a -> b)`

```
const safeHead = (x) => Maybe.of(x[0]);
const increment = (x) => x + 1;
safeHead([1, 2, 3]).map(increment) // -> Just(2)
safeHead([]).map(increment) // -> Nothing
```

```
safeHead = lambda x: Maybe.of(x[0])
increment = lambda x: x + 1
safeHead([1, 2, 3]).map(increment) # -> Just(2)
safeHead([]).map(increment) # -> Nothing
```

- we use `chain`, when we want to apply a function, which uses the fact, that we are working with monad and returns already wrapped value: `(a -> Monad b)`

```
const safeHead = (x) => Maybe.of(x[0]);
const safeDivide = (x) =>
  x === 0
  ? Maybe.Nothing()
  : Maybe.Just(10 / x);
safeHead([1, 2, 3]).map(safeDivide) // -> Just(Just(1))
safeHead([1, 2, 3]).chain(safeDivide) // -> Just(1)
```

```
safeHead = lambda x: Maybe.of(x[0])
safeDivide = lambda x:
  x == 0
  ? Maybe.Nothing()
  : Maybe.Just(10 / x)
safeHead([1, 2, 3]).map(safeDivide) # -> Just(Just(1))
safeHead([1, 2, 3]).chain(safeDivide) # -> Just(1)
```

Monads solve a lot of our problems:

Monad	Use case
Array (List)	working with multiple values
Maybe (Optional)	absence of the value (goodbye `null`, `undefined`, etc.)
Either	error handling (goodbye exceptions)
IO	I/O operations
Task / Lazy Promise	asynchronous operations.
State	shared state inside chain of functions

There is much more:

Applicative functors, Comonads, Transformations, Semigroups, Monoids... most of them are built on top of functors and monads though! We don't have time to dig into anything more though.

Learning resources:

- Mostly Adequate Guide to Functional Programming
- Category Theory for Programmers
- Learn your Haskell
- Try out pure functional languages, like Haskell

Libraries:

- JavaScript and TypeScript: ramda.js, fp-ts
- Python: toolz, pymonad