The background image is a dark, moody landscape. In the foreground, there's a dense forest of tall evergreen trees. Some of these trees have turned a vibrant golden-yellow color, likely due to autumn. The background consists of a dark, rugged mountain slope with visible rock formations and shadows.

# Introduction to functional programming



# Table of contents

1. Motivation
  2. Basic concepts
  3. Higher-order functions
  4. Currying
  5. Composition
  6. Functors and Monads

# 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. very complex or difficult
4. accessible only for people with good understanding of math or only applicable for math-related problems
5. 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, test and verify
4. it's becoming more mainstream (i.e. pattern matching in Python 3.10, promises in ES6)
5. 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:

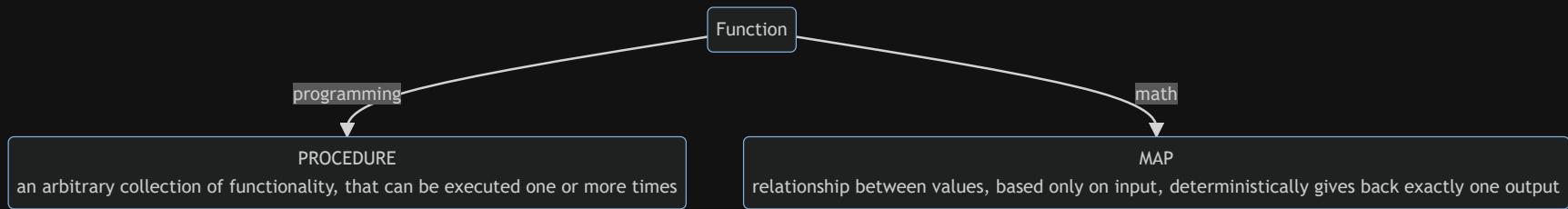
```
function quicksort(array) {
  if (!array.length) {
    return [];
  };
  const [pivot, ...rest] = array;
  const lesser = rest.filter(i => i < pivot);
  const greater = rest.filter(i => i >= pivot);
  return [...quicksort(lesser), pivot, ...quicksort(greater)];
}
```

Haskell:

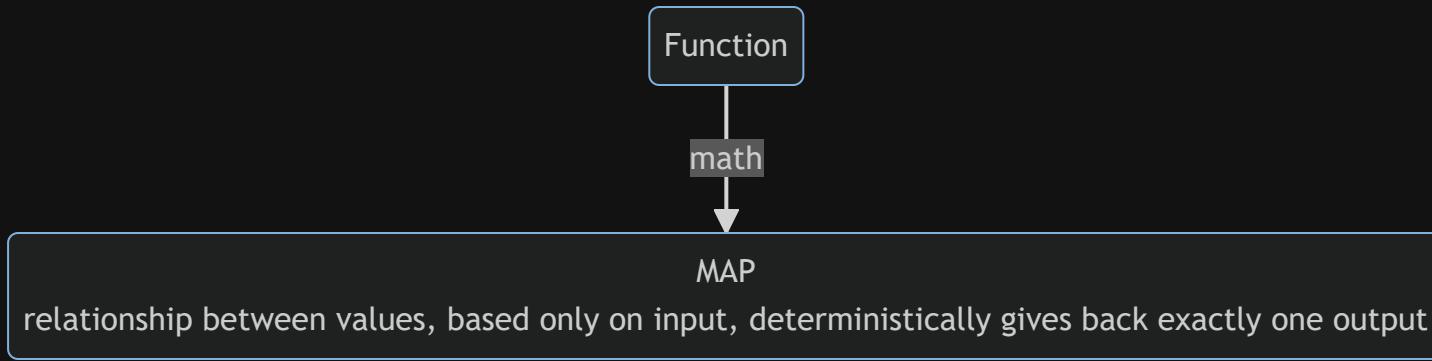
```
quicksort []      = []
quicksort (pivot:rest) = (quicksort lesser) ++ [p] ++ (quicksort greater)
  where
    lesser  = filter (< pivot) rest
    greater = filter (>= pivot) rest
```

# Basic concepts

## Functional programming - programming with functions



## Functional programming - programming with functions



Usually we call these "pure functions".

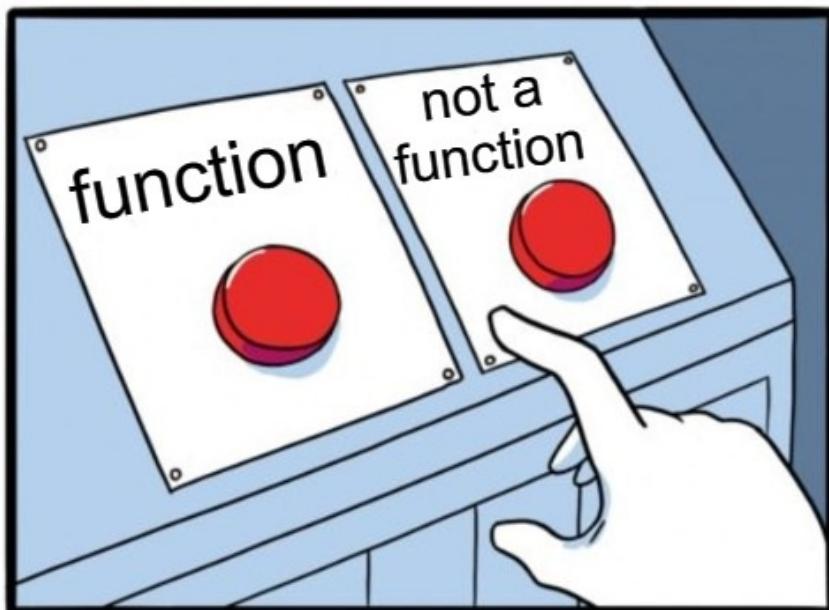
## Implications of this definition:

- 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?



JAKE-CLARK.TUMBLR

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

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

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

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

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

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

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

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

# 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:

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

How it works:

result 0

input data [ 3 , 7 , 31 , 127 ]

# Point free style

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

```
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 its arguments.

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

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

```
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.

```
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...
```

```
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

```
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.

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

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

@curry
def inRange(min, max, val):
    pass

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))`

```
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!

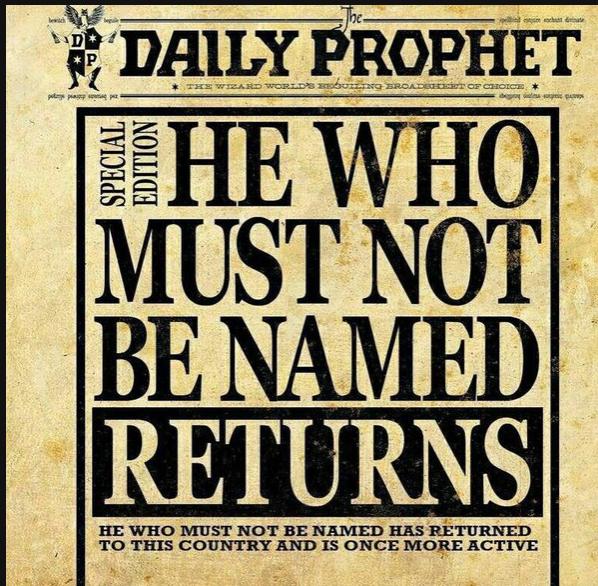
Now we have a very powerful tool, that gives us a very convenient way to "combine" functions!

# 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) => 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

```
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)`

```
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)

```
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)

```
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 <code>null</code> , <code>undefined</code> , 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](#)