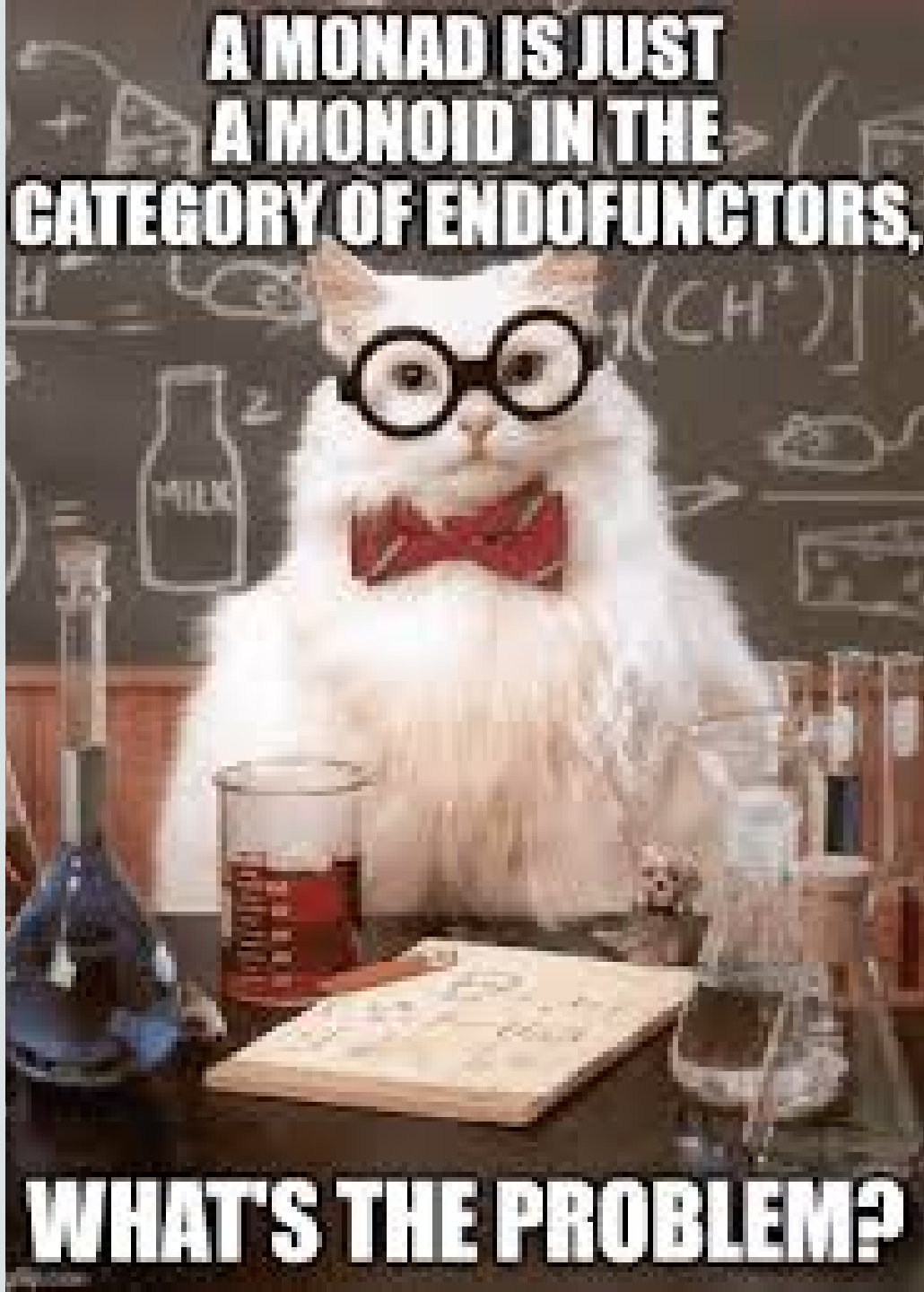
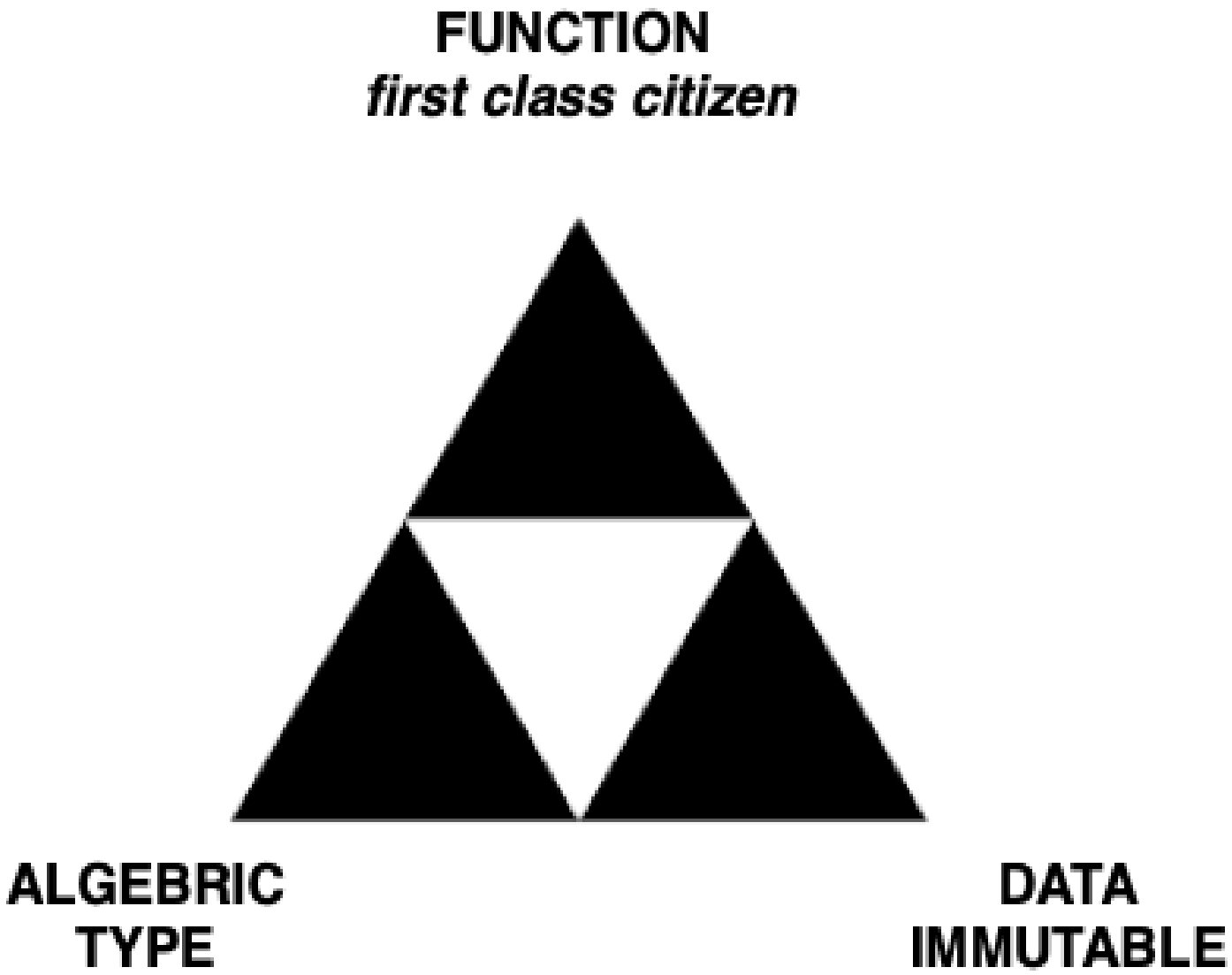
# Monads

Pour du code fonctionnel



# Programmation functionnal



# Function: first class system

> function can be assign to a variable.

```
type GreetFunction = (a: string) => void;
let greet: GreetFunction = (name: string) => {
    console.log("hello, {name}");
}
```

> function can be pass as parameters.

```
function filter1<Type>(arr: Type[], func: (arg: Type) => boolean): Type[] {
  return arr.filter(func);
}
```

# Function: first class citizen

> function can be assign to a variable.

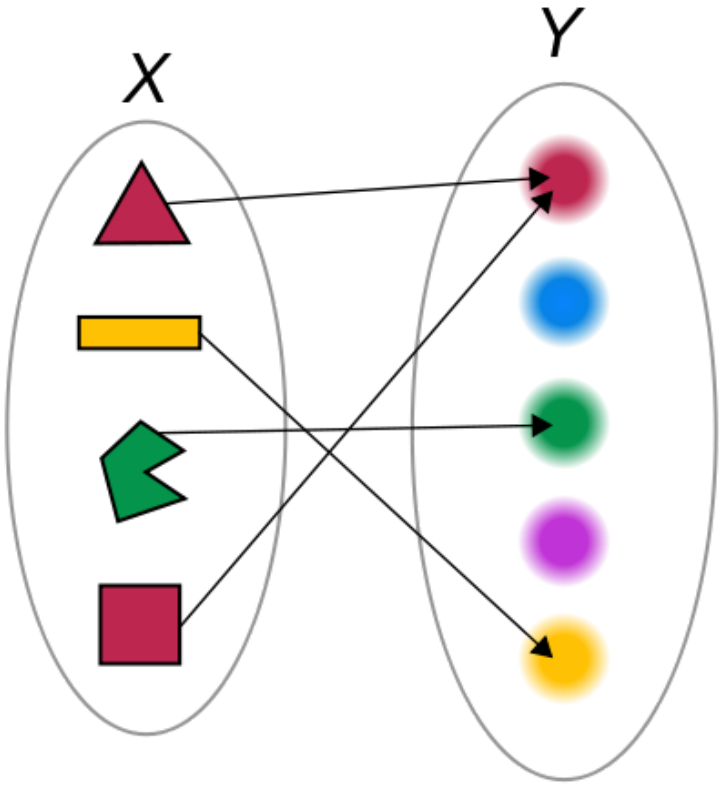
```
type GreetFunction = (a: string) => void;
let greet: GreetFunction = (name: string) => {
    console.log("hello, {name}");
}
```

> function can be return.

```
function middleware(powerBy: string ) => {
    return function (req: Request, res: Response, next: NextFunction): void) {
        res.setHeader('X-Power-By', powerBy);
        next();
    }
}
```

# What is a function ?

> In mathematics, a **function** from a set X to a set Y assigns to each element of X exactly one element of Y.



# Function for programmer

> can produce a error

```
func readFile(path string) (string, error) {...}
```

> can return underterminist value

```
function rand() -> int { return Math.floor(Math.random() * 10); }
```

> can have dependencies

```
let max = 12; let operations = [];
function buffer(value: Any) => void {
    if (operations.length < max) { operations.push(value);}
    else { operations = []; }
}
```

## Function for programmer - pure function

- no variation with local static variables, non-local variables, mutable reference arguments or input streams
- no side effects.

*pure function:*

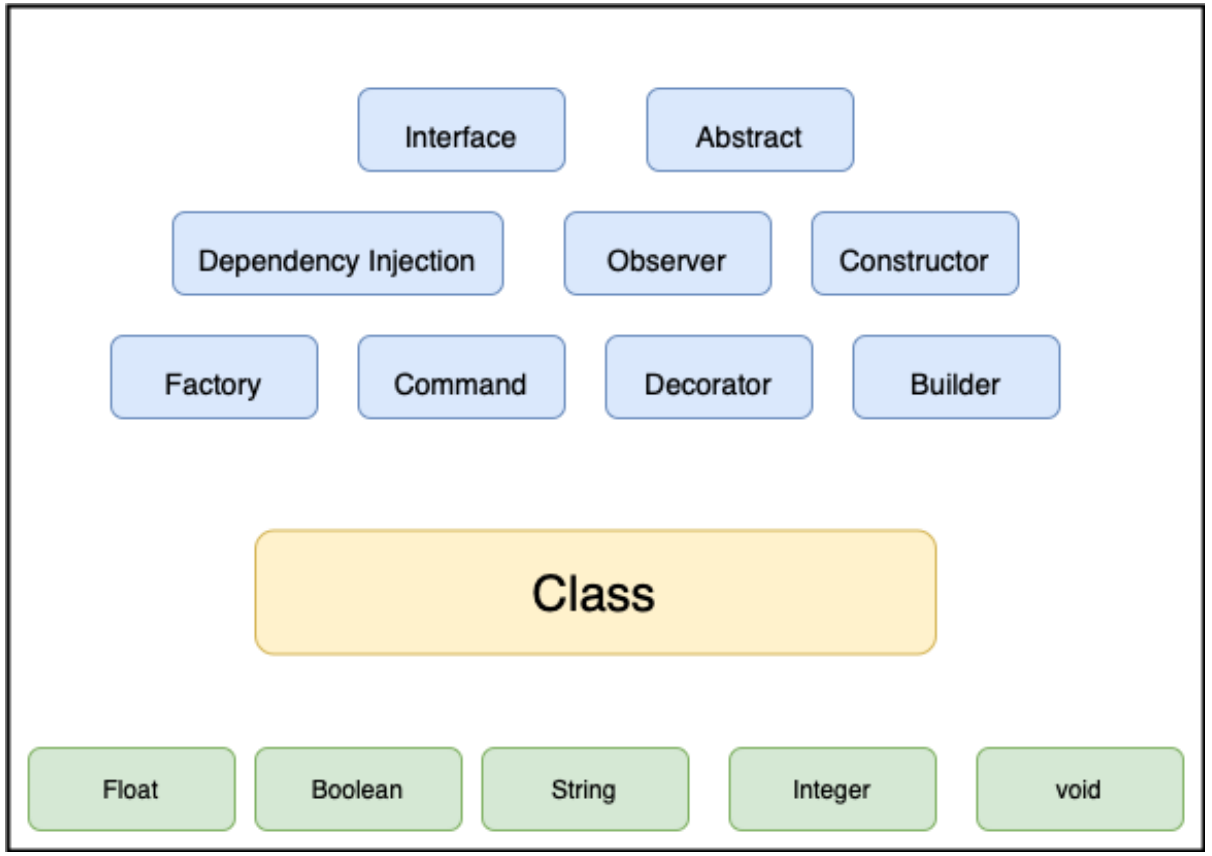
```
void f() {
  static std::atomic<unsigned int> x = 0; ++x;
}
```

*impure function:*

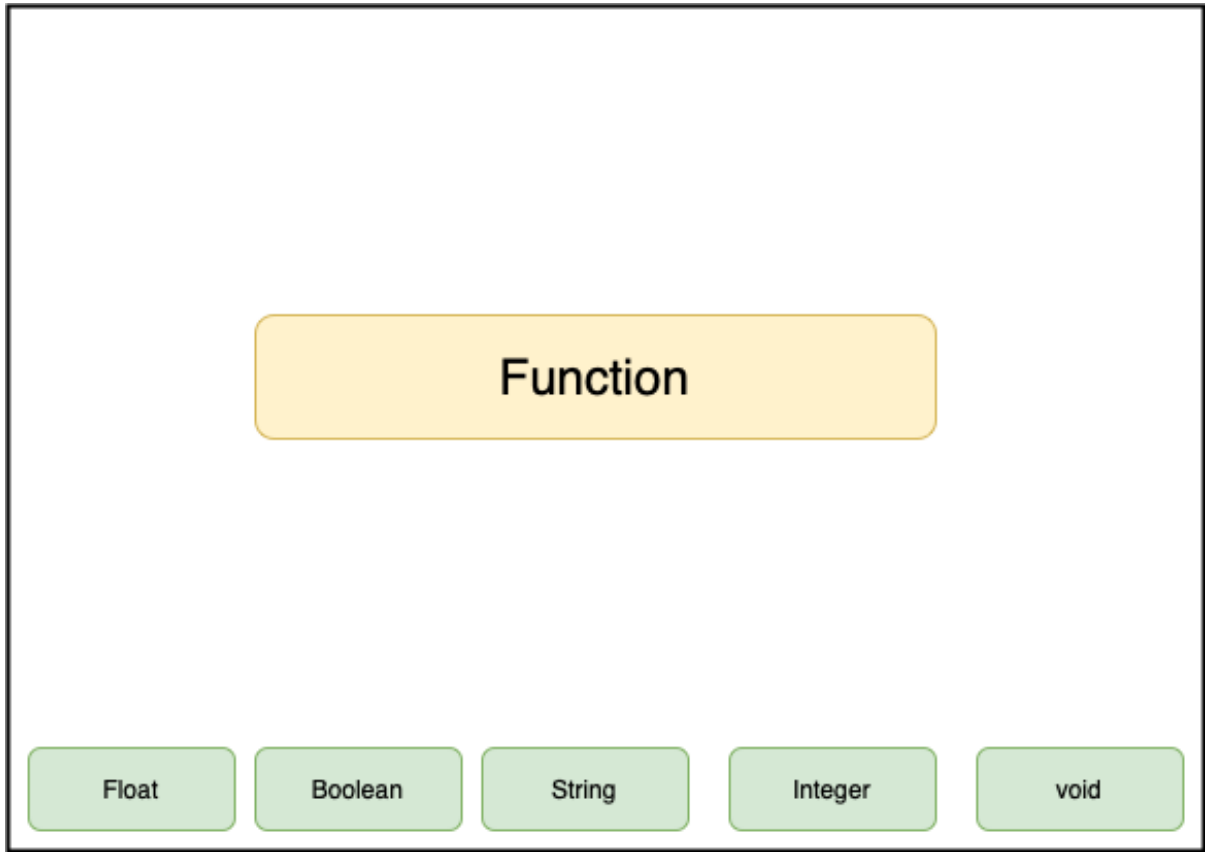
```
int f() {
  static int x = 0; ++x;
  return x;
}
```

# How to organize functional code ?

# Code organization in oop

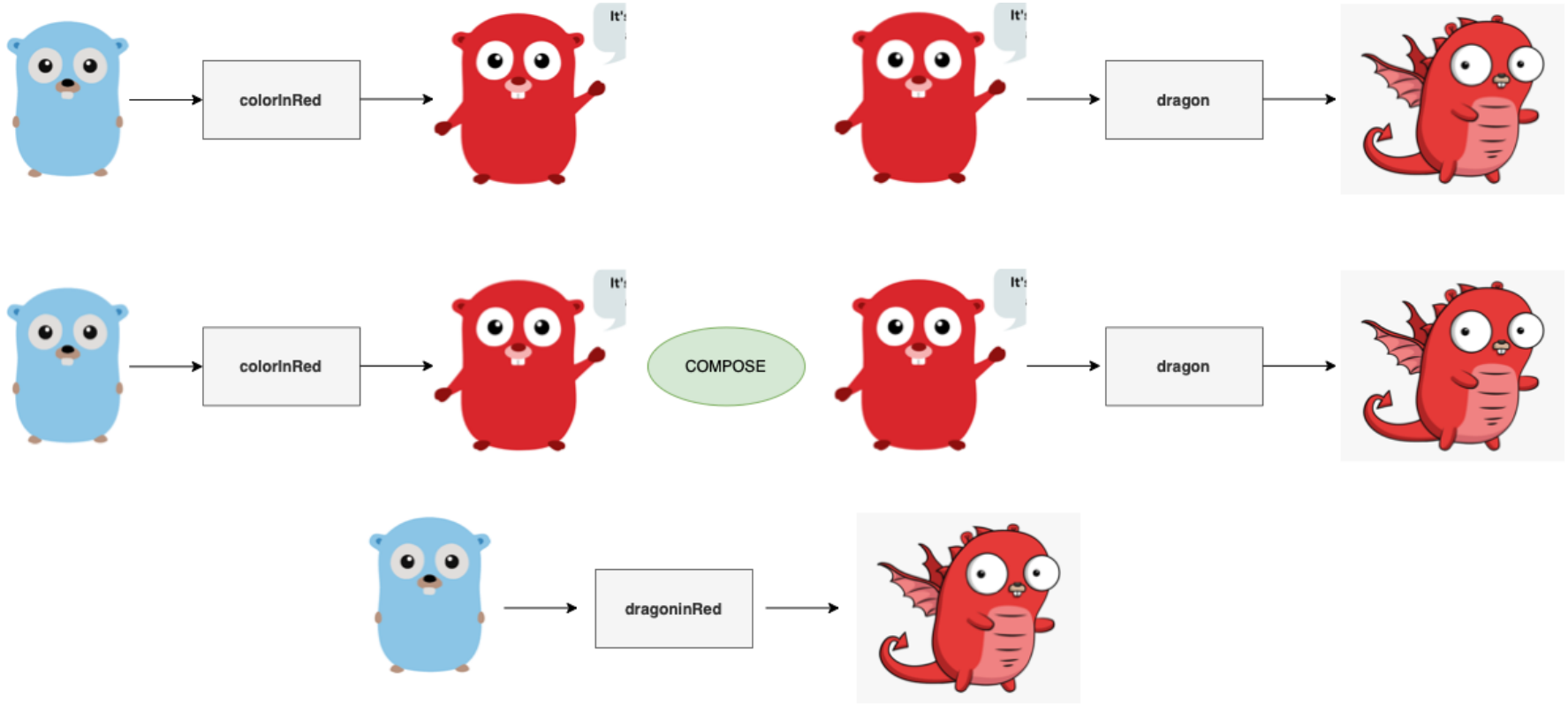


# Code organization in fp



# Composition

# Compostion


colorInRed
dragon
colorInRed
COMPOSE
dragon
dragoninRed

# Compostion in practice

```
function mupl2(i: int) => int { return i * 2; }
function minus4(i: int) => int { return i - 4; }
function square(i: int) => int { return i * i; }
```

> combine function ?

```
function compute(i: int) { // 2*(x^2) - 4
    return minus4(mupl2(square(x)));
}
```

# Compostion in practice


> composition as pipeline

```
compute: Int -> Int
compute =  minus4  . mupl2 . square
```

# Compostion in practice

> example: ***bash pipe***

```
cat file2.txt | sort  | uniq | head -4 > list4.txt
```

> example: ***java stream***

```
List<Float> prices =products.stream()
        .filter(p -> p.price > 30000) // filtering data
        .map(p->p.price)              // fetching price
        .collect(Collectors.toList());
```

> example: ***javascript promise***

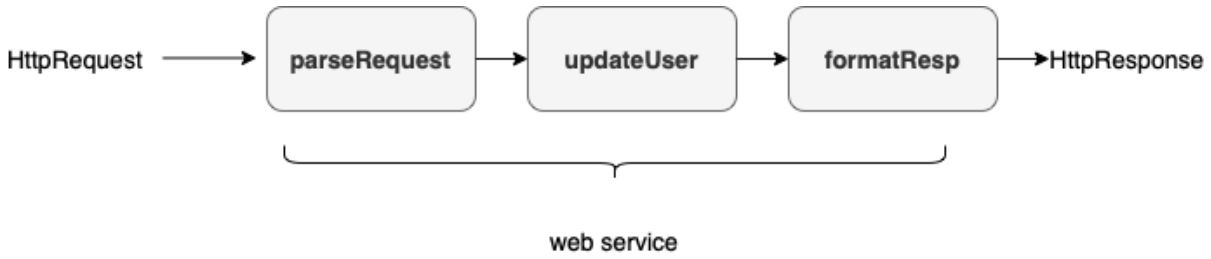
```
Promise.resolve(1).then(2).then(console.log);
```

# Compostion - Example

> Write a web service to update a user's name ?

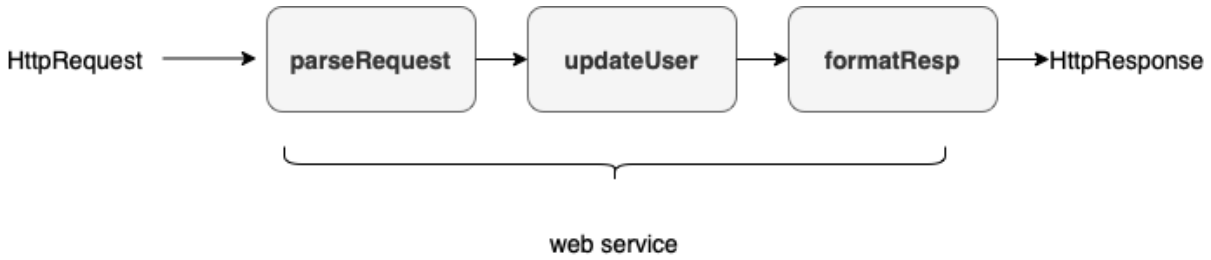
# Compostion - Example

> Write a web service to update a user's name ?



# Compostion - Example

> Write a web service to update a user's name ?




```
handler: HttpRequest -> HttpResponse
compute =  parseRequest  . updateUser . formatResp
```

# Failure

# Function with failure

> the function `parseRequest` could fail
> How to deal with the failure ?


- Ignore - *not a good idea !!!*
- Throw a exception - *where to* `catch` *?*
- Return a `error` when its occurs - *OK*

# Function with failure

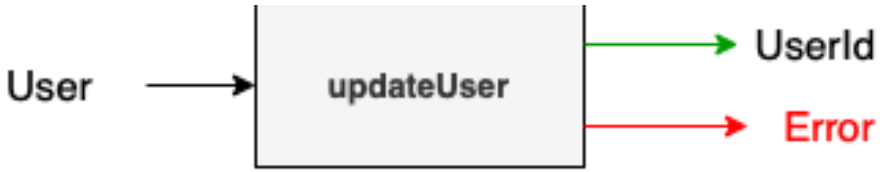

the result of `parseRequest` will be:

- Success - *with the expected result*
- Error - *with the description of why the attemp fail*

```
type Result User =  User | Error
parseRequest :: HttpRquest -> Result User
```

# Function with failure

> the function `parseRequest` and `updateUser` are not longer composable

# Function with failure - Railway programming


- if `parseRequest` fails, return the `error` directly.
- if `parseRequest` succeed, continue the computation.

# Function with failure - Summary

> define a `type constructor`:

```
type Result a =  Either a  Error
```

> define a compostion operator:

```
composeErr :: (a -> Result b) -> (b -> Result c) -> (a -> Result c)
```

`Result` define a `Monad`

# Monad

```
class Monad m where

  (>=>)  :: (a -> m b) -> (b -> m c) -> (a -> m c)

  return ::  a  -> m a
```

# The 4 horsemen of (cat)apocalypse

# The 4 horsemen of (cat)apocalypse

Error

Configuration

Underterminist

External IO

# The 4 horsemen of (cat)apocalypse


~~Error~~

Configuration

Underterminist

External IO

# Function with configuration

> Pure function are self contains.

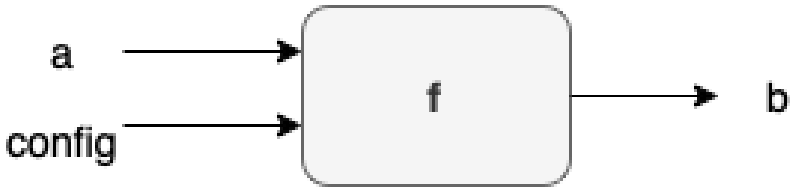
> Computer program are messes of configuration.
>
> Computer program depend on information that is *common knowledge*

We want the ability to pass a *configuration* to every function.

```
type Config = Map String String
funWithConfig :: (a, Config) -> b
```

# Function with configuration- Composition

> What about composition ?




# Function with configuration - curry lemma

> **currying** is the technique of translating the evaluation of a function that takes multiple arguments into evaluating a sequence of functions, each with a single argument.

```
div :: (Int, Int) -> Int

divBy :: Int -> (Int -> Int)
divBy y = \x -> div x y

//conclusion

div x y = (divBy y) $ x
```

# Function with configuration

We want the ability to pass a *configuration* to every function.

```
type Config = Map String String
funWithConfig :: (a, Config) -> b
```

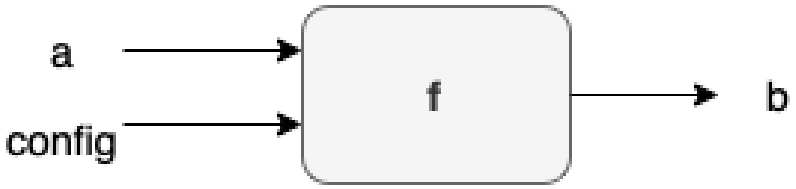
With **currying** operation

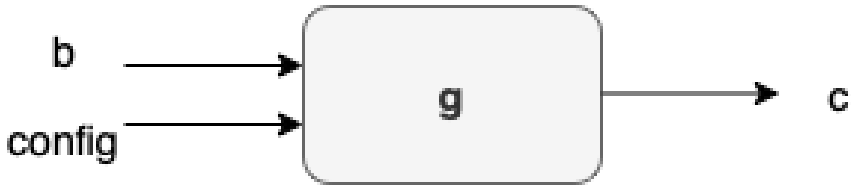
```
funWithConfig :: a -> (Config -> b)
```

```
type Pref b = (Config -> b)
funWithConfig :: a -> Pref b
```

# Function with configuration - Composition

> What about composition ?




- apply the first function with first parameter `a` and `config` to obtain a value of `b`
- apply the second function with the value of `b` and `config` to obtain a value of `c`

# Function with configuration - Summary

> define a `type constructor`:

```
type Pref b = (Config -> b)
```

> define a compostion operator:

```
composePref :: (a -> Pref b) -> (b -> Pref c) -> (a -> Pref c)
```

`Pref` define a `Monad`

# The 4 horsemen of (cat)apocalypse


~~Error~~


~~Configuration~~

Underterminist

External IO

# Uncertain function

> a pure function associate each input to *one* output

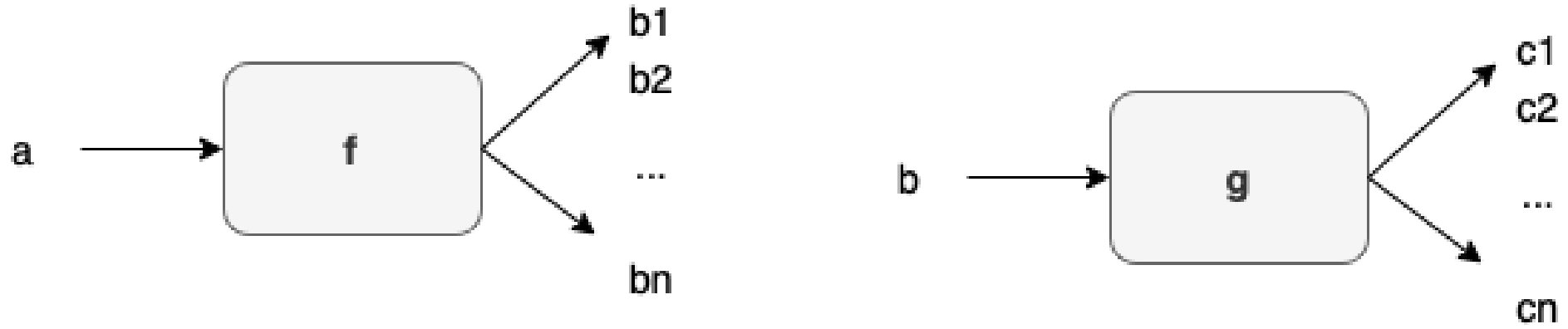
> a non determinist function associate each input to *some number of possibles outputs*.
> eg random, parsing, querying ....

Represent the *possibles outputs* as a `List.`

```
type Possibility b = [b]
nonDeterminist :: a -> Possibility b
```

# Uncertain function - composition

> What about composition ?




- apply the first function `f`.
- apply the seconf function `g` to each output of `f`.

# Uncertain function - summary

> define a `type constructor`:

```
type Possibility b = [b]
```

> define a compostion operator:

```
composePossibility :: (a -> Possibility b) -> (b -> Possibility c) -> (a -> Possibility c)
```

`Possibility` define a `Monad`

# The 4 horsemen of (cat)apocalypse


~~Error~~

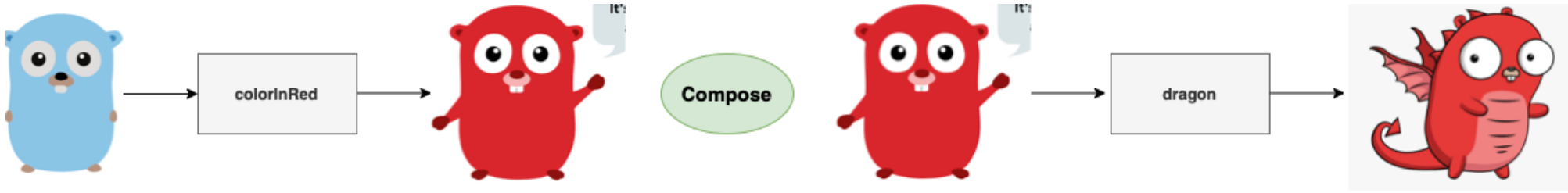

~~Configuration~~


~~Underterminist~~

External IO

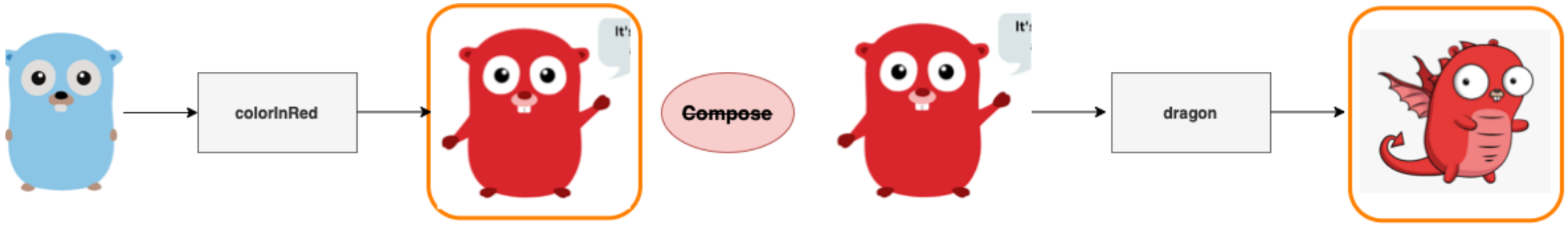
# Summary

# Summary



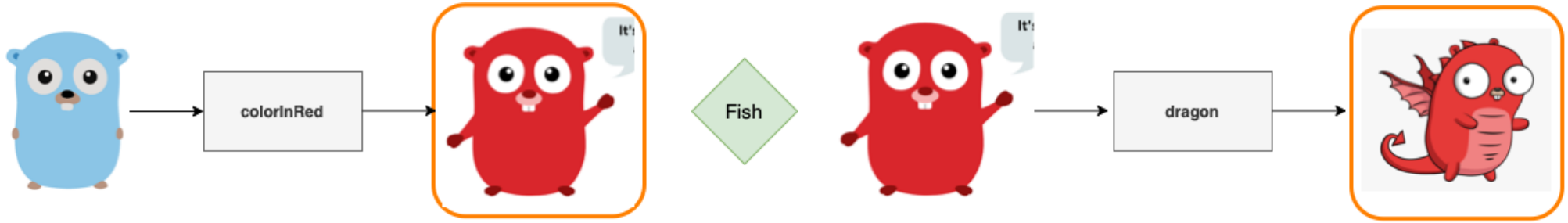

> Let's start with two composable function.
>
> Unfortunately, this functions produce *side-effect*.

# Summary



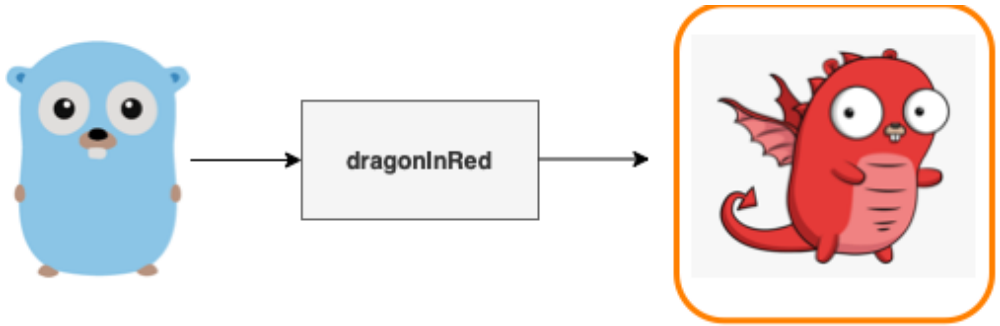

> To deal with side-effect, we need to wrap the return value into a new *type construtor*.
> The functions are not longer *composable*.

# Summary




> To preserve the *composability*, we define a new *operator*.
>
> The *fish* operator.

# Summary




> Function (with side-effect) are compsable again.
>
> The *fish* operator define a *monad*.

Question(s) ?

#41 - ROTI Monads by example