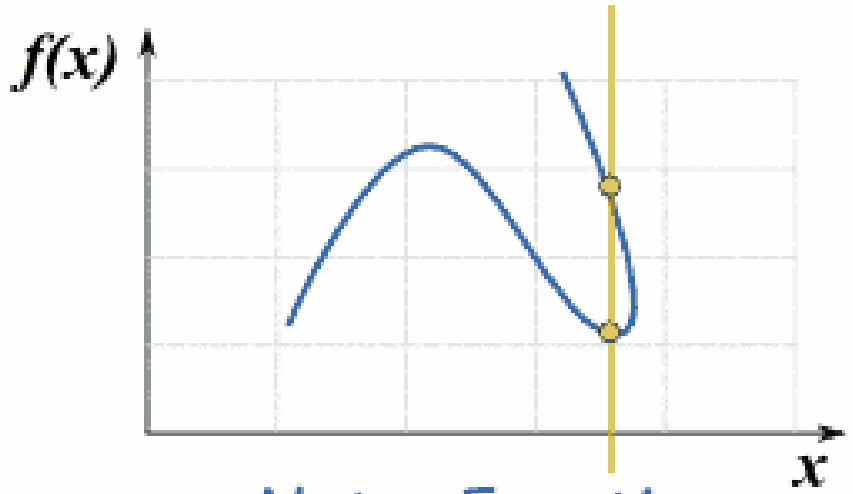
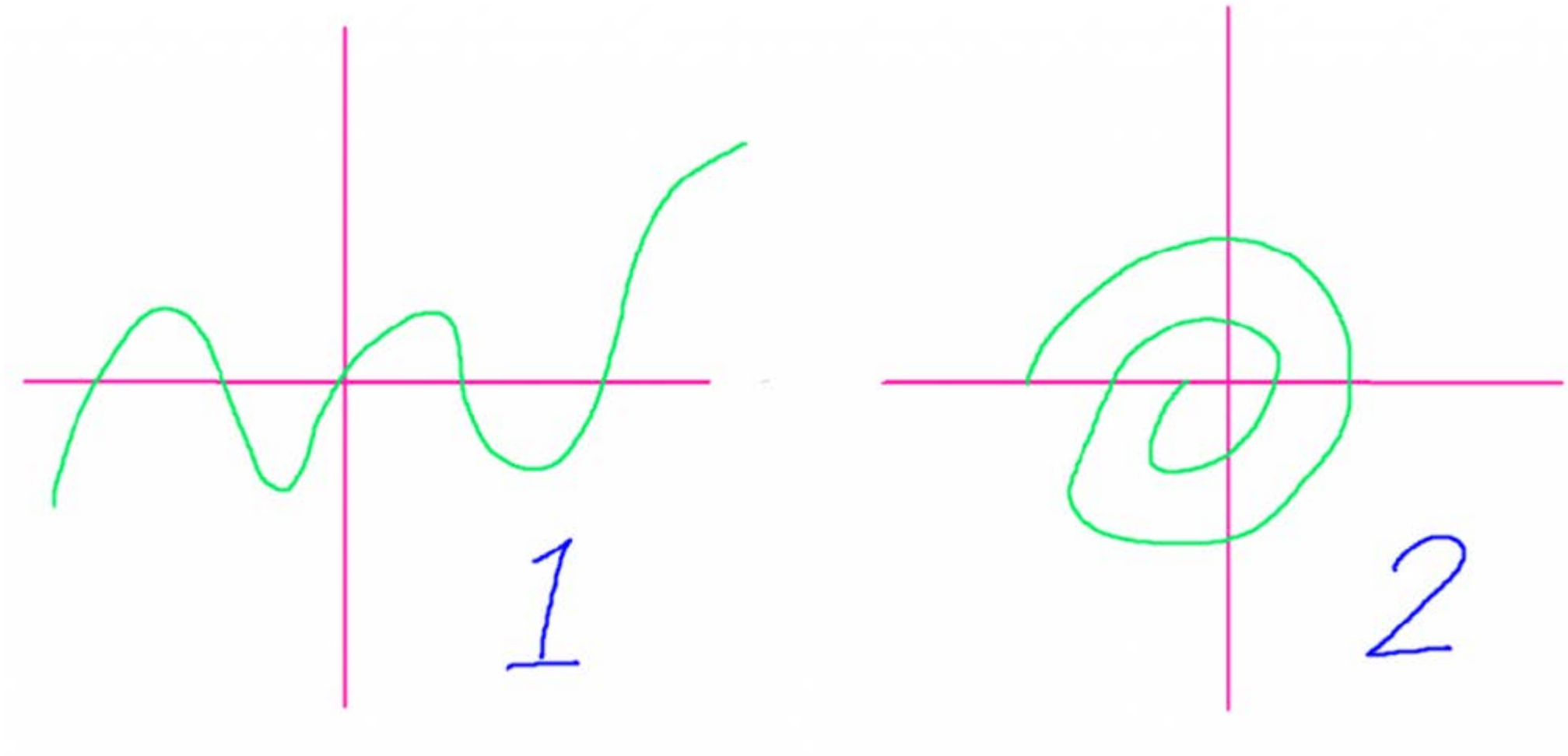
Funtional Programming

Functional ?

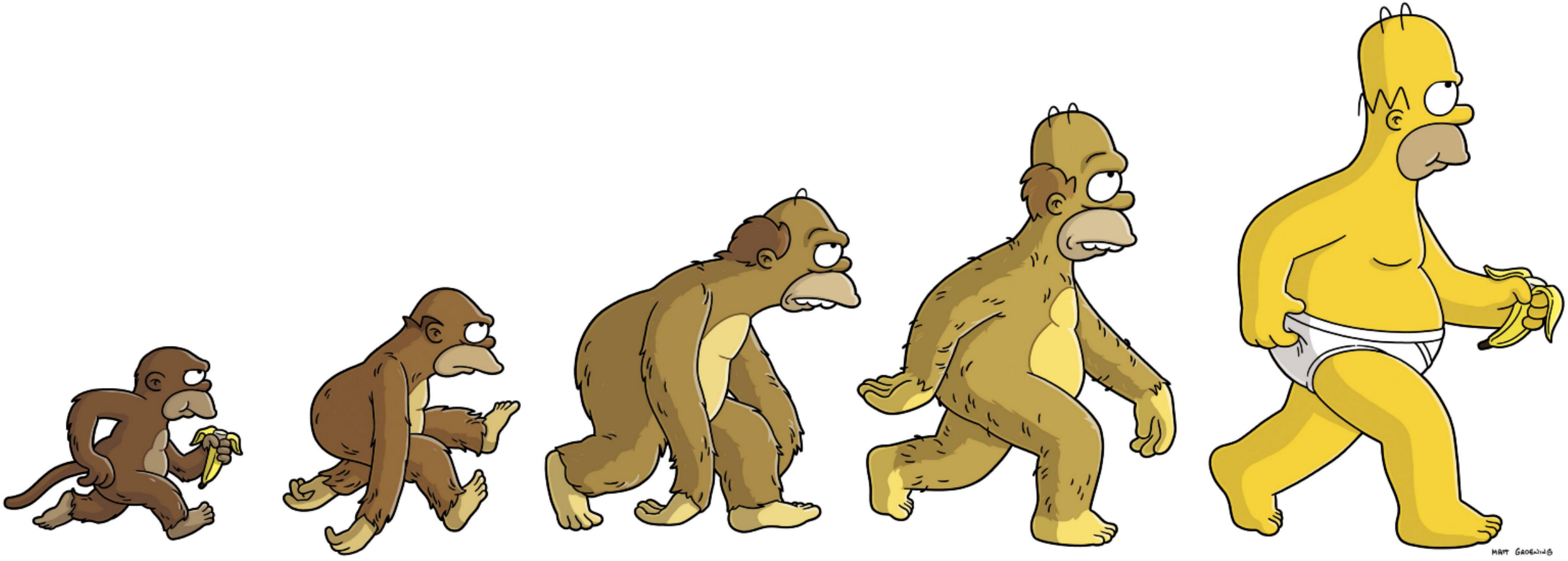
$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{if } x < 0 \end{cases}$$



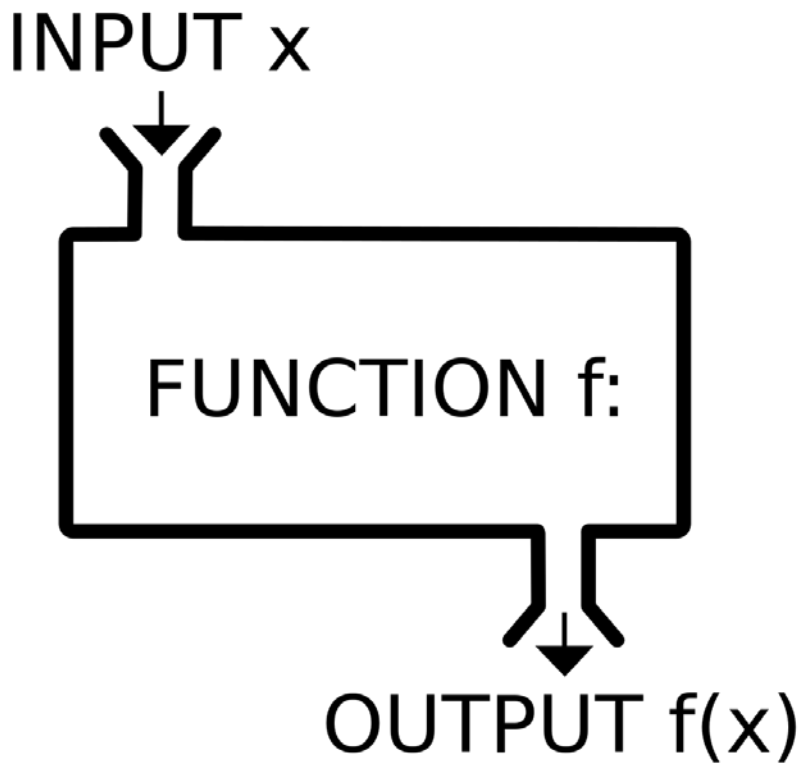
Functional ?

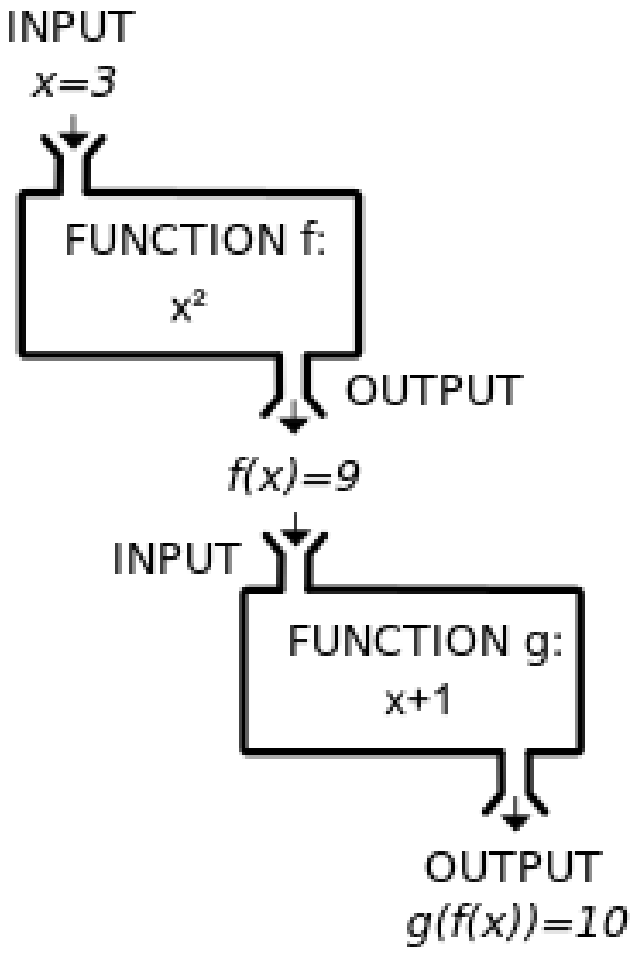


Functional Programming ?



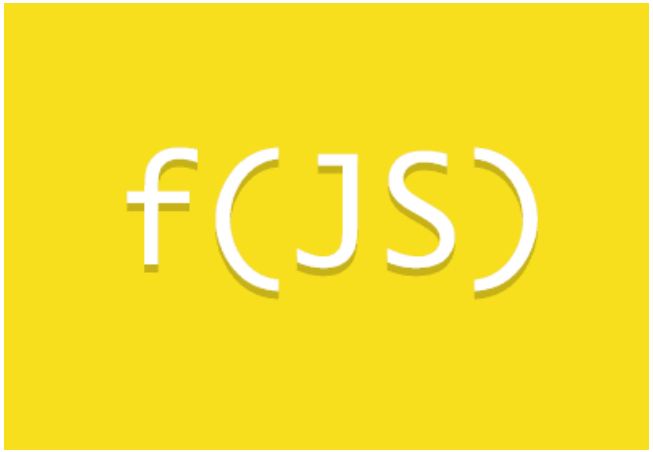
Functional Programming ?





Functional Programming ?

Functional programming (often abbreviated FP) is the process of building software by composing **pure functions**, avoiding **shared state, mutable data,** and **side-effects.** Contrast with object oriented programming, where application state is usually shared with methods in objects.



OO vs Functional

Encapsulation	Pure Function
Abstraction	First Class Function
Inheritance	Immutable Data
Polymorphism	Referential Transparency

Pure Function

- The function always returns the same result if the same arguments are passed in. It does not depend on any state, or data, change during a program's execution. It must only depend on its input arguments.

- The function does not produce any observable side effects such as network requests, input and output devices, or data mutation.

```
function add(a, b){
  return a + b;
}
```

First-Class Function

- A language with first-class functions means that it treats functions like expressions of any other type. Functions are like any other object.

- You can assign a function as a value to a variable:

```
var add = function(a, b){
  return a + b;
}
```

High-Order Function

- Functions that operate on other functions, either by taking them as arguments or by returning them, are called higher-order functions.

```
var add = function(a){
  return function(b){
    return a + b;
  }
}

var add2 = add(2);
var ans = add2(3);
```

Side Effects

- changing the file system
- inserting a record into a database
- making an http call
- mutations
- printing to the screen / logging
- obtaining user input
- querying the DOM
- accessing system state

- If function have a one or more side effect function that it impure function.

Side Effects : Mutation

```
let state = [0, 0];

function pureAdd(arr, a, b) {
  return [arr[0] + a, arr[1] + b];
}

function impureAdd(a, b) {
  state[0]++;
  state[1]++;
  return [state[0] + a, state[1] + b];
}

console.log(pureAdd(state,1,2)); //[1, 2]
console.log(impureAdd(1,2));//[2, 3]
```

Immutable & Mutable

- Mutable Function

```
const numbers = [5, 1, 7, 9, 15, 2]

// Sort จากค่ามากไปหาน้อย
numbers.sort((num1, num2) => num2 - num1)
> [15, 9, 7, 5, 2, 1]

numbers
> [15, 9, 7, 5, 2, 1]
```

```
if (numbers[0] === 5) {
  done()
} else {
  throw new Error("The first element of number must be 5")
}
```

Immutable & Mutable

- Immutable Function

```
const cloneArray = arr => JSON.parse(JSON.stringify(arr))

const sortDesc = (num1, num2) => num2 - num1

const immutableSort = arr => cloneArray(arr).sort(sortDesc)
```

Currying Functional

f(x, y) = $x^2 + y^2$

f(5, 2) = $5^2 + 2^2$

= 25 + 4

= 29

f(x) = f(y) = $x^2 + y^2$

f(5) = f(y) = $5^2 + y^2$

= f(2) = $5^2 + 2^2$

= 25 + 4

= 29

Currying Functional JavaScript

```
function add (a, b) {
  return a + b;
}

add(3, 4); // returns 7
```

```
function add (a) {
  return function (b) {
    return a + b;
  }
}

var add3 = add(3);
add3(4);
```

Currying Functional JavaScript(2)

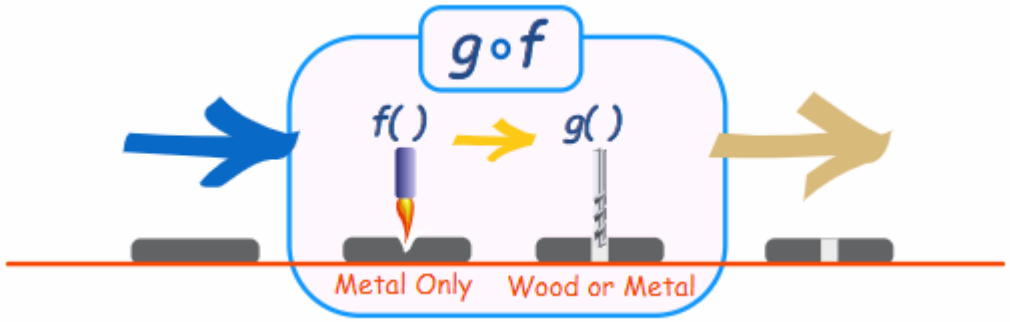
```
multiply = (n, m) => (n * m)

multiply(3, 4) === 12 // true
```

```
curryedMultiply = (n) => ( (m) => multiply(n, m) )

triple = curryedMultiply(3)

triple(4) === 12 // true
```

Compose Function

It is written: (g º f)(x)

Which means: g(f(x))



Example: f(x) = 2x+3 and g(x) = x^2

Compose Function Javascript

```
const compose = function(f, g) {
  return function(x) {
    return f(g(x))
  }
}
```

```
const compose = (f, g) => x => f(g(x))
```

Compose Function Javascript(2)

We will create a function to clean the String before applying it. (Sanitize function)

Step

- Cut off spaces in the front and back of the word.(trim)
- Convert all words into lowercase.(toLowerCase)

```
function sanitize(str) {
  return str.trim().toLowerCase()
}

sanitize(' Hello My Name is Ham ') // 'hello my name is ham'
```

Compose Function Javascript(3)

```
const trim = s => s.trim()
const toLowerCase = s => s.toLowerCase()

function sanitize(str) {
  return toLowerCase(trim(str))
}

sanitize(' Hello My Name is Ham ') // 'hello my name is ham'
```

Analyze

- toLowerCase = f
- trim = g
- str = x
- **toLowerCase(trim(str)) = f(g(x))**

```
const compose = (f, g) => x => f(g(x))

const sanitize = compose(toLowerCase, trim)

sanitize(' Hello My Name is Ham ')
// 'hello my name is ham'
```

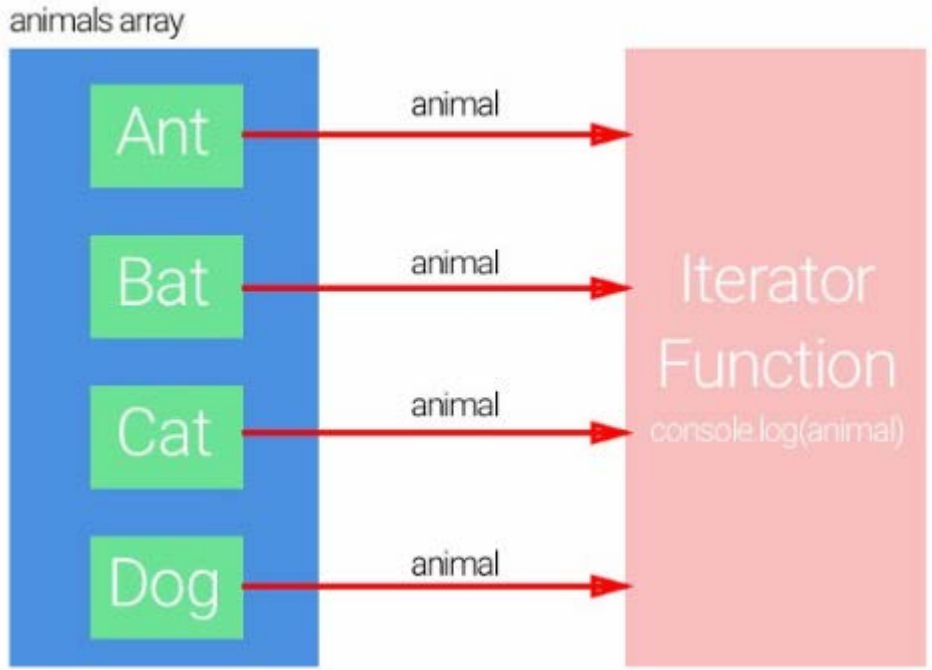
compose is function from lib !!!

Array Method For Functional Programming

- ForEach

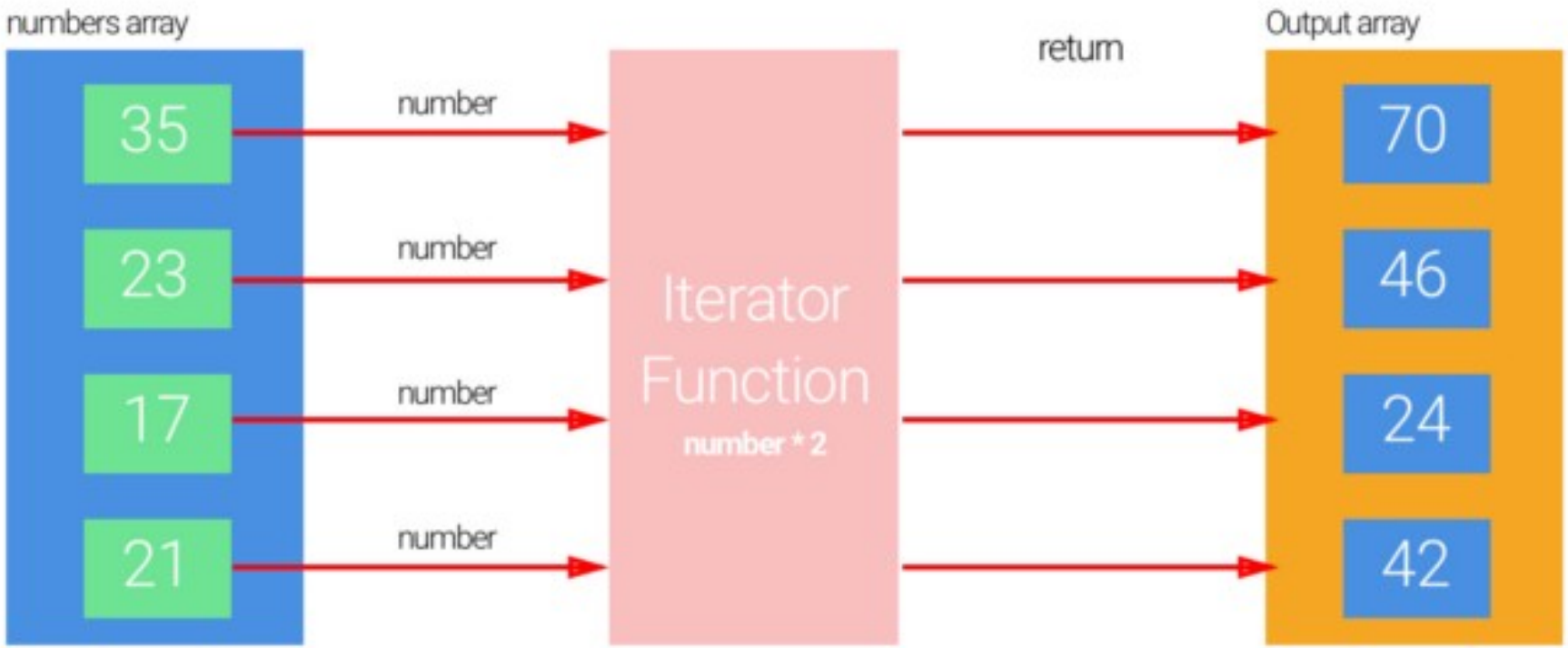
```
const animals = ['Ant', 'Bat', 'Cat', 'Dog']

for (let i = 0; i < animals.length; i++) {
  console.log(animals[i])
}
// Result : "Ant" "Bat" "Cat" "Dog"
```

```
const animals = ['Ant', 'Bat', 'Cat', 'Dog']

animals.forEach((animal) => {
  console.log(animal)
})
// Result : "Ant" "Bat" "Cat" "Dog"
```



Array Method For Functional Programming

- Map



Array Method For Functional Programming

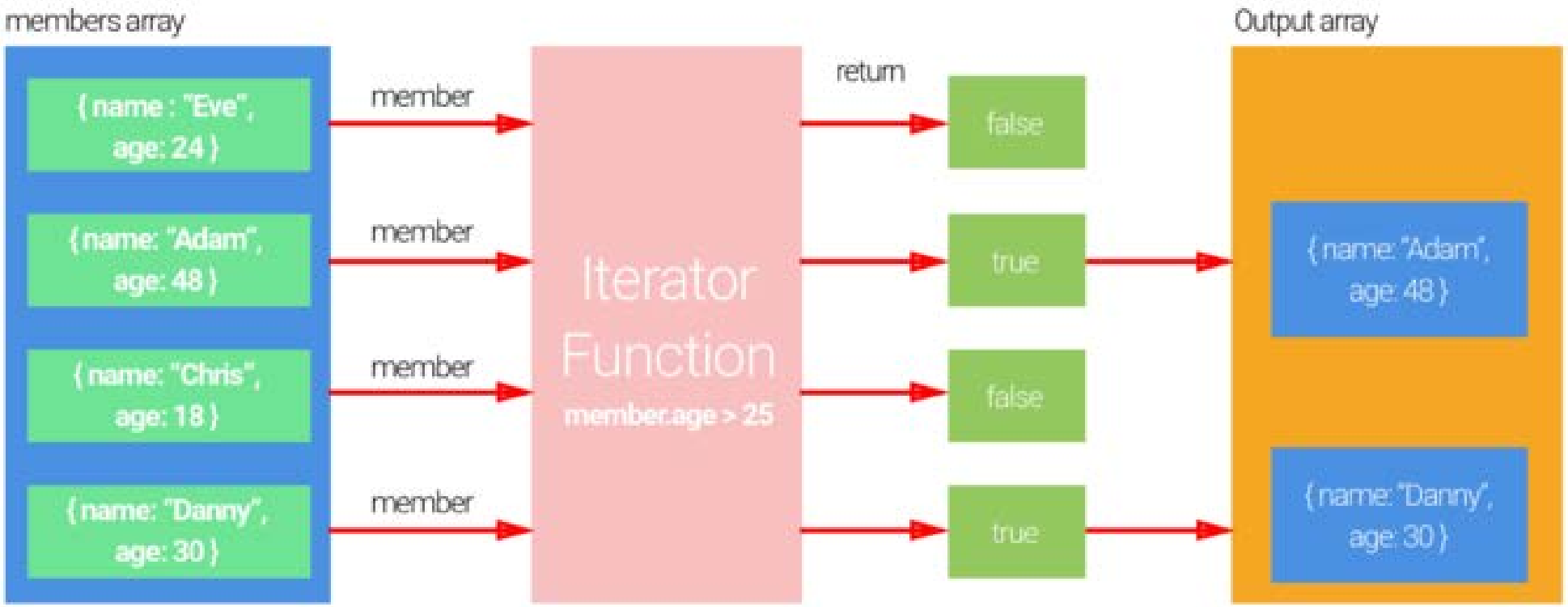
- Map

```
const numbers = [35, 23, 17, 21]

const result = numbers.map((number) => {
  return number*2
})

console.log(result) // [70, 46, 24, 42]
```

```
const numbers = [35, 23, 17, 21]
let result = []

for (let i = 0; i < numbers.length; i++) {
  result.push(numbers[i]*2)
}

console.log(result) // [70, 46, 34, 42]
```

Array Method For Functional Programming

- Filter



Array Method For Functional Programming

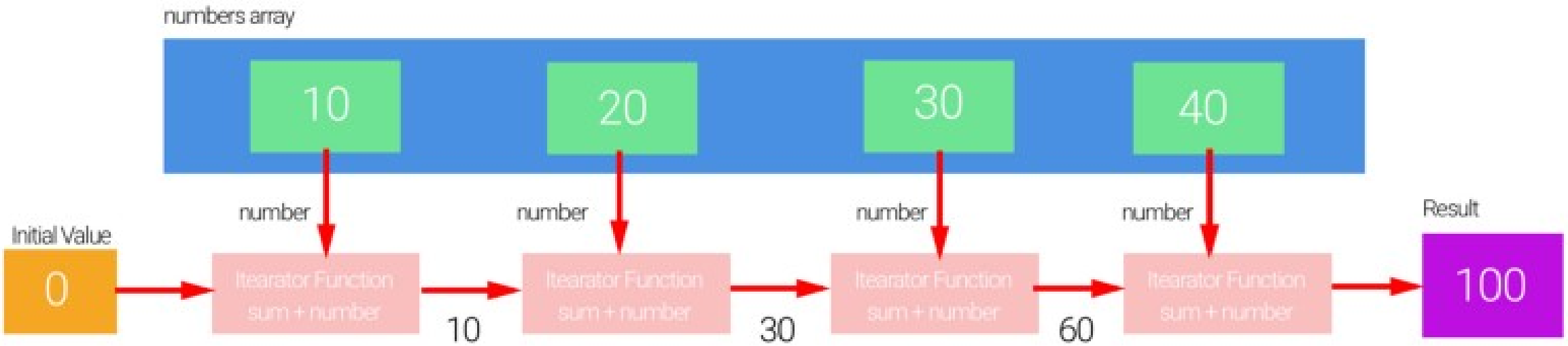
- Filter

```
const members = [
  {name: "Eve", age: 24},
  {name: "Adam", age: 48},
  {name: "Chris", age: 18},
  {name: "Danny", age: 30}
]

const result = members.filter((member) => {
  return member.age > 25
})

console.log(result)
// [{name: "Adam", age: 48}
, {name: "Danny", age: 30}]
```

```
const members = [
  {name: "Eve", age: 24},
  {name: "Adam", age: 48},
  {name: "Chris", age: 18},
  {name: "Danny", age: 30}
]
let result = []

for (let i = 0; i < members.length; i++) {
  if (members[i].age > 25) {
    result.push(members[i])
  }
}

console.log(result)
// [{name: "Adam", age: 48}
, {name: "Danny", age: 30}]
```

Array Method For Functional Programming

- Reduce



Array Method For Functional Programming

- Reduce

```
const numbers = [10, 20, 30, 40]

const result = numbers.reduce((sum,number) => {
  return sum+number
}, 0)

console.log(result) // 100
```

```
const numbers = [10, 20, 30, 40]

let result = 0

for (let i = 0; i < numbers.length; i++) {
  result += numbers[i]
}
console.log(result) // 100
```


LIBRARY


Central Library, King Mongkut's Institute of Technology La...
Jaturapat Thaiwong, Insight360plus
ดูภายใน - มี.ค. 2016
สำนักหอสมุดกลาง
กลับไปที่แผนที่
Google
เวลาถ่ายภาพ: มี.ค. 2016
© 2016 Google
ข้อกำหนด
ส่วนบุคคล
maps.google.co.th
รายงานปัญหา

RAMDA

compose

Function

`((y → z), (x → y), …, (o → p), ((a, b, …, n) → o)) → ((a, b, …, n) → z)`

EXPAND PARAMETERS

Added in v0.1.0

Performs right-to-left function composition. The rightmost function may have any arity; the remaining functions must be unary.

Note: The result of compose is not automatically curried.

See also pipe.

Open in REPL Run it here

```
var classyGreeting = (firstName, lastName) => "The name's " + lastName + ", " + firstName + " " + lastName
var yellGreeting = R.compose(R.toUpper, classyGreeting);
yellGreeting('James', 'Bond'); //=> "THE NAME'S BOND, JAMES BOND"

R.compose(Math.abs, R.add(1), R.multiply(2))(-4) //=> 7
```

curry

Function

`(* → a) → (* → a)`

EXPAND PARAMETERS

Added in v0.1.0

Returns a curried equivalent of the provided function. The curried function has two unusual capabilities. First, its arguments needn't be provided one at a time. If `f` is a ternary function and `g` is `R.curry(f)`, the following are equivalent:

- `g(1)(2)(3)`
- `g(1)(2, 3)`
- `g(1, 2)(3)`
- `g(1, 2, 3)`

Secondly, the special placeholder value `R.__` may be used to specify "gaps", allowing partial application of any combination of arguments, regardless of their positions. If `g` is as above and `_` is `R.__`, the following are equivalent:

- `g(1, 2, 3)`
- `g(_, 2, 3)(1)`
- `g(_, _, 3)(1)(2)`
- `g(_, _, 3)(1, 2)`
- `g(_, 2)(1)(3)`
- `g(_, 2)(1, 3)`
- `g(_, 2)(_, 3)(1)`

See also curryN.

Open in REPL Run it here

```
var addFourNumbers = (a, b, c, d) => a + b + c + d;

var curriedAddFourNumbers = R.curry(addFourNumbers);
var f = curriedAddFourNumbers(1, 2);
var g = f(3);
g(4); //=> 10
```

Reference

- http://ramdajs.com/docs/#curry
- http://ramdajs.com/docs/#compose
- **Learning React: Functional Web Development with React and Redux** 1st Edition, Kindle Edition