

Programmation fonctionnelle en JavaScript :



Igor Laborie
Expert Web & Java

 @ilaborie

 igor@monkeypatch.io



Langages pratiqués

1. Java
2. JavaScript (TypeScript, CoffeeScript)
3. Kotlin, Scala

Notions dans

Python, SML, Racket, Rust, Swift, Go, ...



- I. Langages fonctionnels
- II. Programmation fonctionnelle en JS - Part I
- III. Entracte
- IV. Programmation fonctionnelle en JS - Part II
- V. Remarques sur la performance
- VI. Conclusion

LANGAGES FONCTIONNELS

Fooling around with alternating current (AC) is just a waste of time.
Nobody will use it, ever. -- Edison, 1889

*

“

There is no reason anyone would want a computer in their home. --
Ken Olson, 1977

“

• I predict the Internet will soon go spectacularly supernova and in

- C est un langage fonctionnel
- JavaScript est un langage fonctionnel
- Java est un langage OO
- JQuery est une monade

Paradigmes

- programmation impérative
- programmation orientée objet
- programmation fonctionnelle
- ...

“

On peut adopter donc un style de programmation fonctionnelle avec la plupart des langages. Les caractéristiques des langages peuvent rendre cela plus ou moins facile (voir obligatoire)

C'est quoi un langage fonctionnel ?

“

Il n'y a qu'un langage fonctionnel : le λ -calcul

developpeur fonctionnel => Devel

(Haskell, normaux, humour, paresseux, lambda)

Programmation fonctionnelle 4/5 Typeage statique

Dynamique

Lisp (1958)

Racket (1994), Clojure (2007), ...

Statique

ML (1973)

Haskell (1990), Scala (2004)

PROGRAMMATION FONCTIONNELLE EN JS - PART I

Functions en JavaScript



```
// Function is First-class citizen
```

```
function mult(a, b) {  
    return a * b;  
}
```

```
console.log(`Plop`);
```

```
// variable
```

Source: [https://www.youtube.com/watch?v=...](#)

```
var sum = 0;  
  
[1, 2, 3, 4, 5]  
  .forEach(elt => sum += elt);  
console.log({sum});
```

⚠ danger, c'est un nid à bugs.

=> Éviter les fonctions qui n'ont pas de paramètres, ou retournent `void`

```
const sum = [1, 2, 3, 4, 5]  
  .reduce((acc, elt) => acc += elt);
```

referentially transparent

avec  ImmutableJS

Comment fait-on ?

```
class List<T> {  
    private array: T[];  
  
    constructor(elements: T[] = []) {  
        this.array = [... elements];  
    }  
  
    add(element: T) : List<T> {
```

High Order function

map, reduce, filter, sort

```
// f: X  $\Rightarrow$  Y  
// g: Y  $\Rightarrow$  Z  
// compose: (Y  $\Rightarrow$  Z, X  $\Rightarrow$  Y)  $\Rightarrow$  X  $\Rightarrow$  Z  
const compose = (g, f)  $\Rightarrow$  x  $\Rightarrow$  g(f(x))
```



```
const factorial3 = (n, acc = 1) =>
  (n ≤ 1) ? acc : factorial3(n - 1, acc * );
```



Support de la tail-rec)

⚠ Éviter les recursions entre plusieurs fonctions:

```
const a = n => {
  if (n > 42) {
    return b(n - 42);
  }
  return 2;
}
```

```
speakers
  .filter(speaker => speaker.xp > 10 &&
    speaker.some(lang => lang === 'JavaScript'))
```

```
speakers
  .filter(speaker => speaker.xp > 10) // is experimented
  .filter(speaker => speaker.some(lang => lang === 'JavaScript'))
```

```
const isExperimented = speaker => speaker.xp > 10;
const isLoveJS = speaker => speaker.loves.some(lang => lang === 'JavaScript');
```

```
speakers
  .filter(isExperimented)
  .filter(isLoveJS)
```



- immutable faisable ✍️
- éviter les effets de bord 🧳
 - Boucles: `while`, `do ... while`, `for`, `for ... of`,
`for ... in`
 - les `var` ou `let` => utiliser `const`
 - les fonctions sans arguments, ou qui
 - les mutations dans les objects comme `o.x = 5;` =>
utiliser `Object.assign` ou la déconstruction
 - les mutations dans Array : `copyWithin`, `fill`, `pop`,
`push`, `reverse`, `shift`, `sort`, `splice`, `unshift`
 - les mutations dans Map : `clear`, `delete`, `set`

ENTRACTE


By Adrian Pingstone (Photographed by Adrian Pingstone) [Public domain],  via [Wikimedia Commons](#)

Photo by Lance Anderson on Unsplash

Ailurus fulgens

Habitat: the Himalayan forests of Nepal and China.

PROGRAMMATION FONCTIONNELLE EN JS - PART II

A functor is a collection of X that can apply a function $f : X \rightarrow Y$ over itself to create a collection of Y .

(Array with map)

Reduction

Function: $(X, X) \rightarrow X$

Combinators

Composition

into a chain of functions of one argument that will yield the same result when called in sequence with the same arguments.

$$f(x, y, z) = g(x)(y)(z)$$

```
const mult = (a: number, b: number) => a * b;
```

```
// idea: identity = mult(1, _)
const identity = (a: number) => mult(1, a);
```

```
// better
const multCurry = (a: number) => {
  return (b: number) => a * b;
}
```


High Order Function

➡ <https://medium.freecodecamp.org/understanding-memoize-in-javascript-51d07d19430e> ➡
<https://www.sitepoint.com/implementing-memoization-in-javascript/>

```
const fibonacci = function(n) {  
  switch (n) {  
    case 1 : return 1;  
    case 2 : return 1;  
    default:  
      return fibonacci(n-2) + fibonacci(n-1);  
  }  
}
```

Abstract Data Type

```
datatype rational = Whole of int  
                  | Frac   of int*int
```

TypeScript

type union

```
let { x, y, z } : vector = Math.sqrt(x ** 2 + y ** 2 + z ** 2)
    { x, y } : vector = Math.sqrt(x ** 2 + y ** 2),
    [ ... ] : vector.length,
    else: {
        throw new Error("Unknown vector type");
    }
}
```

Pour implémenter du pattern-matching il faut de la déconstruction. On l'a déjà sur les {} et les [].

```
const myPoint = { x: 14, y: 3 };
const {x, y} = myPoint; // x ≡ 14, y ≡ 3

const tab = [1, 2, 3, 4];
```

```
Some(1)  
  .map { it + 1 }  
  .flatMap { Some("4" + it) }  
  .map { it }
```

- langage souple permet pas mal de manipulation
- manque `flatMap`
- mais ➡ Ramda

REMARKS SUR LA PERFORMANCE

Performance en quoi ?

- temps d'exécution (mimimum, maximun, moyen, première exécution) ?
- consomation de mémoire ?
- consomation d'énergie ?

“

Douter de toutes les mythes et légende

- on fait aux bonnes structures de données (Data oriented design)
- on évite les IO (disque, réseau), c'est l'occasion de faire de la FRP
- le code doit être bien testé
- on privilégie la lisibilité du code à une (hypothétique) optimisation de performance
- mettre en cache n'est pas toujours la bonne solution

- si besoin de meilleures performances (à définir), on définit le seuil désiré
- on effectue des mesures
- on isole la zone à optimiser, idéalement la plus petite possible
- on commente, pour expliquer pourquoi on n'a perdu de la lisibilité
- on suit l'évolution des performances dans toute la durée de vie du projet

CONCLUSION

- les bases sont présentes.
- écosystème dans ce domaine, plutôt à la mode

Mais encore insuffisant à mon goût:

- flatMap 😭  Proposal for flatten and flatMap - Stage 3
- un lazy Array (comme les Stream Java)

“

Program designers have a tendency to think of the users as idiots who need to be controlled. They should rather think of their program as a servant, whose master, the user, should be able to control it. If designers and programmers think about the apparent mental qualities that their programs will have, they'll create programs that are easier and pleasanter — more humane — to deal with.

-- John McCarthy "The Little Thoughts of Thinking Machines", Psychology Today, December 1983, pp. 46–49.
Reprinted in Formalizing Common Sense: Papers By John McCarthy, 1990, ISBN 0893915351

“

La valeur principale d'un programme est dans ce qu'il réalise, pas dans son style.

Mais en temps qu'artisan développeur, j'accorde de l'importance au style.

expressivité souplesse ecosystème évolué dans le bon sens

- moins de bugs
- plus simple
- plus évolutif *



What Is Wrong With Mutability?

- Hard to reason (prone error)
- Hard to make concurrent or parallelize

What Is Wrong With side-effect function?

- Hard to reason (prone error)
- Hard to make concurrent or parallelize

slides

-  tc39 ECMAScript proposals
-  Fantasy Land Specification (aka "Algebraic JavaScript Specification")
- TODO eslint rules: immutable, pure, ...

Questions ?