Programmation fonctionnelle en JavaScript :

 ou  ?

Igor Laborie
Expert Web & Java

@ilaborie

igor@monkeypatch.io

<Monkey Patch/>

Langages pratiqués

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

Notions dans

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

Plan

I.  Langages fonctionnels

II. Programmation fonctionnelle en JS - Part I

III. Programmation fonctionnelle en JS - Part II

IV.  Remaques sur la performance

V. Conclusion

VI. Quizz

Langages fonctionnels

> Fooling around with alternating current (AC) is just a waste of time. Nobody will use it, ever.
>
> *Edison, 1889 inventeur, scientifique, fondateur de General Electric*

> There is no reason anyone would want a computer in their home.
>
> *Ken Olson, 1977 cofondateur DEC*

> I predict the Internet will soon go spectacularly supernova and in 1996 catastrophically collapse.
>
> *Robert Metcalfe, 1995 inventeur ethernet, fondateur de 3com*

Catharsis

> C est un langage fonctionnel

> `JavaScript` est un langage fonctionnel

> `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 carcactéristiques des langages peuvent rendre cela plus ou moins facile (voir obligatoire)

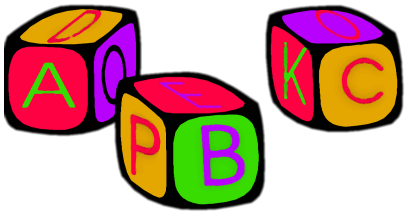
One to rule them all

Mais alors, c'est quoi un langage fonctionnel ?

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

Fellowship

Developpeur fonctionnel



Developpeur impératif ou OO

Photo by Vasilios

Statique vs Dynamique

> Programation fonctionnelle ⇏ Typage statique

Typage dynamique		Typage statique	
Lisp	(1958)	ML	(1973)
Scheme	(1975)	Haskell	(1990)
Racket	(1994)	OCaml	(1996)
Clojure	(2007)	Scala	(2004)
...		...	

Programmation fonctionnelle en JS - Part I

Functions en JavaScript

```ts
// Function is First-class citizen
function mult(a, b) {
    return a * b;
}

console.log(typeof mult); // function
console.log(mult(3, 14));

// Variable
const mult2 = function(a, b) {
    return a * b;
};

// ES2015+
const mult3 = (a, b) => a * b;

// TypeScript
const mult4 = (a: number, b: number): number => a * b;
```

Effets de bord


```ts
let sum = 0;

[1, 2, 3, 4, 5]
    .forEach(elt ⇒ {
        sum += elt
    });

console.log({sum});
```

- ⚠️ Évitez les fonctions avec effet de bord !
- C'est un nid à bugs.
- ⇒ Évitez les fonctions qui retournent `void`, ou qui n'ont pas de paramètres.

Functions sans effet de bord

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

console.log({ sum });
```

Instruction vs Expression

```ts
const isEven = (n: number): boolean =>
    (n % 2 === 0);

[2, 3, 4, 5]
    .forEach(elt => console.log(elt, 'even?', isEven(elt)));
```

Mess with state

> The last thing you wanted any programmer to do is mess with internal state even if presented figuratively. Instead, the objects should be presented as *sites of higher level behaviors more appropriate for use as dynamic components.*

--Alan Key, The Early History Of Smalltalk

Immutable

```ts
// let & const
let oLet = {a: 1, b: 2};
oLet.a = 3;


// Transparence référentielle ?
const oConst = Object.freeze({a: 10, b: 20});
oConst.a = 30;

// Deconstruction
const oConst2 = { ...oConst, a: 30};

console.info({oLet});
console.info({oConst});
console.info({oConst2});
```

Immutable

Comment fait-on avec les structures de données ?

```ts
class List<T> {
    private array: T[];

    constructor(elements: T[] = []) {
        this.array = [ ... elements];
    }

    add(element: T) : List<T> {
        return new List([ ... this.array, element]);
    }
}
```

On peut utiliser Immutable.js , mori

High Order functions

```ts
const digits = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
type Predicate<T> = (T) => boolean;


const isEven = (n: number): boolean => (n % 2 === 0);

const not = <T>(fun: Predicate<T>): Predicate<T> =>
    (n: T) => !fun(n);

const isOdd = not(isEven);


const evenDigits = digits.filter(isEven);

const oddSquared = digits
    .filter(isOdd)
    .map(n => n ** 2);

console.log(evenDigits);
console.log(oddSquared);
```

for : 🤢

#15

```ts
interface Event { error: boolean; /* ... */}

const funErrors = (events: Event[], size = 10): Event[] =>
    events
        .filter(evt => evt.error)
        .slice(0, size);

const notFunErrors = (events: Event[], size = 10): Event[] => {
    const result: Event[] = [];
    for (let i = 0; i < events.length; i++) { // 🤮
        const evt = events[i];
        if (evt.error) {
            result.push(evt);
        }
        if (result.length >= size) {
            return result;
        }
    }
    return result;
};
```

Composition


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

const composeAll = ( ...functions) =>
    x => functions.reduceRight((value, fun) => fun(value), x);
```

TC39 Stage 1 - pipeline operator

```js
const doubleSay = (str) => str + ', ' + str;
const capitalize = (str) => str[0].toUpperCase() + str.substring(1);
const exclaim = (str) => str + '!';

const result1 = exclaim(capitalize(doubleSay('hello')));

const result2 = 'hello'
    |> doubleSay
    |> capitalize
    |> exclaim;
```

Lisibilité


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

```js
speakers.filter(speaker => speaker.xp > 10) // is experimented
        // is love JS
        .filter(speaker => speaker.languages.some(lang => lang === 'J
```

```js
const isExperimented = speaker => speaker.xp > 10;
const isLoveJS = speaker => speaker.languages.some(lang => lang === '

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

```js
speakers.filter(and(isExperimented, isLoveJS));
```

Part I - bilan 1/2

- 💪 `function` first-class citizen
- ✍️ immutable faisable
  - `Object.freeze`, `Object.seal`
  - Stage 1 - `Object.freeze` + `Object.seal` syntax
  - ⇒ bibliothèques
- ⚠️ eviter les effets de bord ⇒ 💖 tests

  fonctions sans arguments, ou retournant `void`
- TC39 Stage 1 - `do` expressions

Part I - bilan 2/2

- ✋ `while`, `do ... while`, `for`, `for ... of`, `for ... in`
- ✋ `var` ou `let` ⇒ ❤️ `const`
- ✋ `o.x = 5;` ⇒ ❤️ `Object.assign` ou `{ ... o, x: 5}`
- ✋  mutateurs Array: `push`, `pop`, `shift`, `unshift`, `sort`, `splice`, ...
- ❤️  accesseurs Array: `filter`, `map`, `slice`, `reduce` ...
- ✋ Map : `clear`, `delete`, `set`
- ✋ Set : `add`, `clear`, `delete`

Programmation fonctionnelle en JS - Part II

Ce qu'on a appris

- Function as First Class Citizen
- High Order Function
- Referential Transparency
- Idempotent

Ce qu'on va voir maintenant

- Curryfication,
- Memoïsation,
- Algebraic Data Type,
- Pattern Matching,
- Functor, Monïd, Monad, ...

Curryfication

```ts
// type: number ⇒ number ⇒ number
const mult = a ⇒ b ⇒ a * b;

const identity = mult(1);

const double = mult(2);

const triple = mult(3);

[identity, double, triple]
    .map(fun ⇒ fun(42))
    .forEach(x ⇒ console.log(x));
```

Curryfication

> Transformation d'une fonction de plusieurs arguments en une chaîne de fonctions d'un seul argument qui donnera le même résultat lorsqu'il est appelé en séquence avec les mêmes arguments.

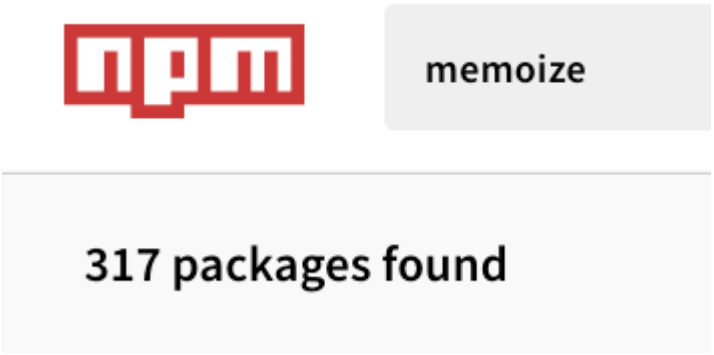
```
f(x, y, z) = g(x)(y)(z)
```

- Viens de Ⓦ Moses Schönfinkel et Ⓦ Haskell Curry
- TC39 Stage 1 - Partial Application Syntax
- Ramda curry
- `Function.bind`
- ⚠ sens des arguments

```typescript
type IntFun = (number) => number;

const stupidMemoizer = (fun: IntFun): IntFun => {
    const cache: number[] = [];
    return (n: number) => {
        const cached = cache[n];
        if (typeof cached === 'number') {
            return cached;
        }
        return (cache[n] = fun.call(null, n));
    }
};

const fibonacci: IntFun = stupidMemoizer(n => {
    switch (n) {
        case 1 :
            return 1;
        case 2 :
            return 1;
        default:
            return fibonacci(n - 2) + fibonacci(n - 1);
    }
});

console.log('fibonacci(15)', fibonacci(15));
```

Memoïsation

- ➳ _.memoize
- 💎 purtée ⟹ on peut mettre en cache !



Algebraic Data Type


```
type schoolPerson = Teacher
                  | Director
                  | Student(string);
```

RE

😞 les *enum* ou les *type union* de TypeScript ne sont pas des ADT.

adt.js

JS

Pattern Matching


```
let greeting = stranger =>
  switch (stranger) {
  | Teacher => "Hey professor!"
  | Director => "Hello director."
  | Student("Richard") => "Still here Ricky?"
  | Student(anyOtherName) => "Hey, " ++ anyOtherName ++ "."
  };
```

RE

```
const getLength = vector => {
    match (vector) {
        when { x, y, z } ~> return Math.sqrt(x ** 2 + y ** 2 + z ** 
        when { x, y } ~> return Math.sqrt(x ** 2 + y ** 2)
        when [ ...etc] ~> return vector.length
    }
}
getLength({x: 1, y: 2, z: 3}) // 3.74165
```

JS

🔮 TC39 Stage 0 - ECMAScript Pattern Matching

JS

Déconstruction


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

const tab = [1, 2, 3, 4];
const [head, ...tail] = tab; // head == 1, tail == [ 2, 3, 4]
```

M-word

> A monad is just a monoïd in the category of endofunctors, what's the problem?
>
> --🤡

Functor

> Généralisation aux catégories de la notion de morphisme.
>
> --😂

```ts
interface Functor<A> {
    map(mapper: (A) => B): Functor<B>;
}
```

Fantasy Land `Functor`

```ts
interface EndoFunctor<V> {
    map(mapper: (V) => V): Functor<V>;
}
```

Monoïd

> C'est un magma associatif et unifère, c'est-à-dire un demi-groupe unifère.
>
> --🤣

```ts
interface SemiGroup {
    concat: (SemiGroup) ⇒ SemiGroup;
    // this.concat(x.concat(y)) = this.concat(x).concat(y)
}
```

```ts
interface Monoid extends SemiGroup {
    empty: Monoid;
    // monoid.concat(empty) = monoid, empty.concat(monoid) = monoid
}
```

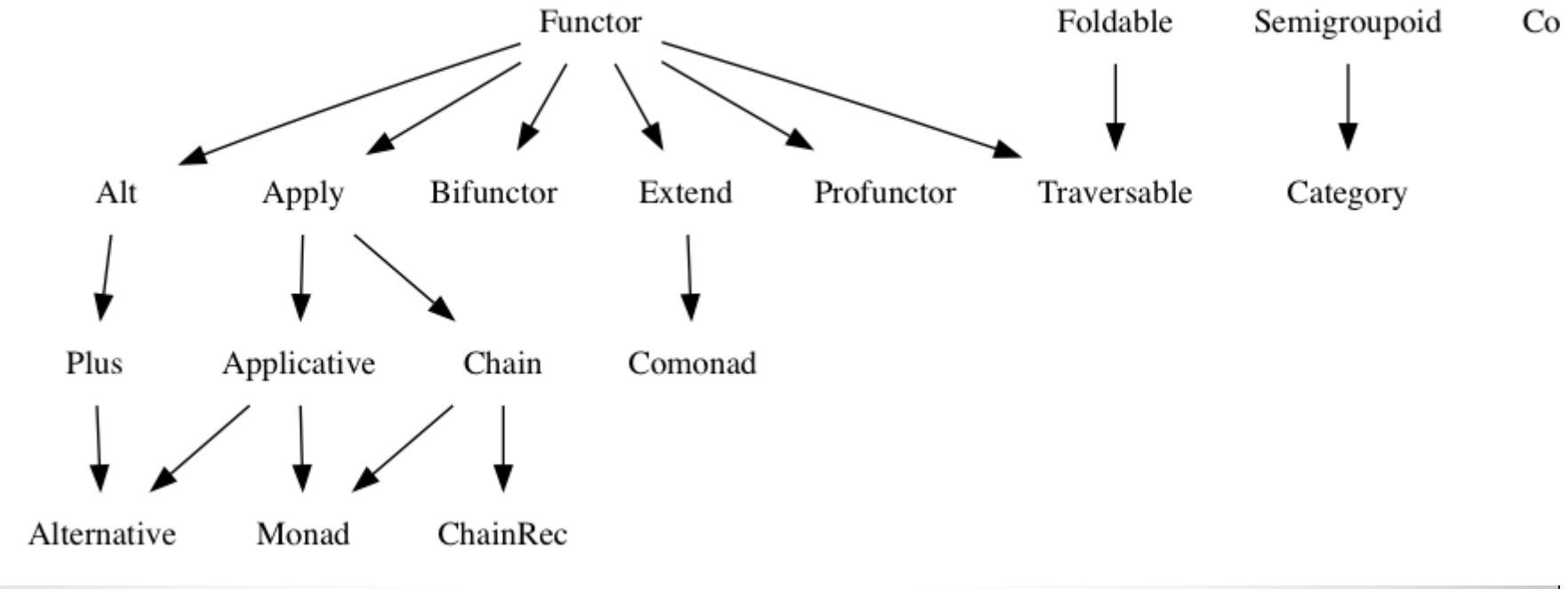
Fantasy Land **Monoid** , Fantasy Land **Semigroup**

Monade


```ts
interface Monad<A> extends Functor<A> {
    flatMap(mapper: (A) ⇒ Monad<B>): Monad<B>;
}
```

TC39 Stage 3 - `Array.prototype.{flatMap,flatten}`

Fantasy Land `Monad`



Monade pour les humains

> J'ai toujours pas compris !
>
> --😳, 🤯

- C'est un ~~objet~~ 🌯
- qui a des méthodes simples comme par exemple `map` ou `flatMap`
- qui doivent respectées des règles (axioms)
- ce qui garenti une haute composabilité.
- `Option<V>`, `Either<A,B>`, `Result<S,E>`, `Future<V>`, ...

Monades en JS

- ça ressemble aux `Promise<T>`
- Pour comprendre correctement les monades, il faut comprendre la Théorie des catégories ! Ou pas ?

- Fantasy Land - Algebraic JavaScript Specification
- Practical Intro to Monads in JavaScript
- Javascript Functor, Applicative, Monads in pictures

Part II - bilan

- langage souple permet pas mal de manipulation
- manque `flatMap` dans les api
- on peut utiliser  Ramda,  monet.js, Folktale, ...
- Ne pas se sentir exclu de la FP à cause du jargon !

Remaques sur la performance

Quoi ?

Performance en quoi ?

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

> Douter de toutes les mythes et légendes

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

Si besoin...

> tout les leviers sont bon, y compris le langage

- on définit le seuil désiré
- on effectue des mesures
- on isole la zone à optimiser (la plus petite possible)
- on commente pourquoi on n'a perdu de la lisibilité
- on suit l'évolution des performances

Conclusion

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

- 😭 `flatMap`
- des structures lazy (comme les `Stream` de Scala)
  - 😡 il n'y a pas d'emoji pour les paresseux !

    ➳ Proposal for SLOTH Emoji

Bibliothèques

- ECMAScript Proposals
- Immutable.js, mori: ClojureScript's persistent data
- Ramda, monet.js, Folktale, adt.js, FantasyLand compliant (monadic) alternative to Promises
- eslint-plugin-fp, eslint-config-cleanjs
- ...
- Awesome FP JS

Alternatives

- Elm
- ReasonML
- ClojureScript
- PureScript
- …

Intéret du JS

- 🗣️ expressivité
- 🤸 souplesse
- 💰 ecosystème
- 🦎 évolution dans le bon sens

Codons en fonctionel

- 🍼 plus simple
- ✅ Plus facile à tester
- 🐛 moins de bugs
- 🦎 plus évolutif
- ♻️ applicable sur tous les (bon) langages
- 🎓 apprendre

> 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.

Liens

🧛Slides HTML: http://bit.ly/funJS

Questions ?