

Programmation fonctionnelle

Avancé

Developper Fullstack Javascript

Fullstack developer for about fourteen years, I like making software in all its aspects: user needs, prototypes, UI, frontend developments, API developments, ...

Team & Projects


I work on DataClient teams where we work for tooling around data client and perform the data client quality.

Hobbies


I like drawing, UX,
Web Design




SAMUEL GOMEZ

 samuel-gomez

 @gamuez

 www.samuelgomez.fr

 gamuez_art

Séance 1 : Base de JavaScript

Rappel rapide des bases.

Séance 2 : JavaScript avancé

Manipulation du DOM, appels HTTP, nouveauté ES6, ...

Séance 3 : Programmation fonctionnelle (bases)

Les concepts de bases.

Séance 4 : Programmation fonctionnelle (avancée)

Les concepts plus avancés.

Séance 5 : Les bases de React

JSX, composants, ...

Séance 6 : Gestion d'état

Etat local, Context API, Fetch, ...

Currying

- Le principe
- Les librairies
- Exercices
- Corrigés

Point Free Style

- Le principe
- Exercices
- Corrigés

Recursion

- Le principe
- La pile d'exécution
- La TCO
- Exercices
- Corrigés

Functor

- Le principe
- Exemple : Array
- Exercices
- Corrigés

Currying

Currying

Le principe

En PF, la curryfication désigne la transformation d'une fonction à plusieurs arguments en une fonction à un argument qui retourne une fonction sur le reste des arguments (Wikipedia)

Arité

En [mathématiques](#), l'arité d'une [fonction](#), ou [opération](#), est le nombre d'arguments ou d'[opérandes](#) qu'elle requiert. (Wikipedia)

```
const mult = (a, b) => a * b;  
// si on souhaite multiplier par 2 les éléments du tableau  
const values = [1, 2, 3];  
  
console.log('doubles', values.map((item) => mult(item, 2)));  
console.log('triple', values.map((item) => mult(item, 3)));
```

```
// fonctionne seulement pour 2 paramètres  
const curry = fn => {  
  return function(a) {  
    return function(b) {  
      return fn(a,b)  
    }  
  }  
}  
  
// raccourcie  
const curry2 = fn => a => b => fn(a,b);  
console.log('curry double', values.map(curry2(mult)(2)));  
console.log('curry triple', values.map(curry2(mult)(3)));
```

Currying

Les libraries

Dans l'exemple précédent, la fonction `curry` permet d'avoir 2 éléments en entrée, ce qui peut vite être limitant. On pourrait réécrire une fonction `curry` plus générique qui ressemblerait à ça :

```
function curry(fn, arity = fn.length) {  
  return (function nextCurried(prevArgs) {  
    return function curried(nextArg) {  
      let args = [...prevArgs, nextArg];  
  
      // tant que l'arité n'est pas atteinte, retourne une nouvelle fonction  
      // qui récupérera le (ou les) paramètres suivants  
      if (args.length >= arity) return fn(...args);  
  
      // sinon, on appelle la fonction finale avec tous les arguments  
      else return nextCurried(args);  
    };  
  })([]);  
}
```

Mais il existe des librairies comme [Lodash](https://lodash.com/docs/4.17.15#curry) ou [Ramda](https://ramdajs.com/docs/#curry) qui nous évite de devoir réécrire à chaque fois de genre de fonction.

<https://lodash.com/docs/4.17.15#curry>

<https://ramdajs.com/docs/#curry>

Currying Exercices



<https://codesandbox.io/s/exo-currying-ezib7>



Point Free Style

Point Free Style

Le principe

La programmation tacite, ou **point free style** est un style d'écriture des fonctions où, lors de l'appel de fonction, on ne spécifie pas explicitement les arguments sur lesquels la fonction travaille. Pour illustrer cela, reprenons notre exemple des notes d'étudiants, nous voudrions convertir les notes sur 20.

```
const multiply = a => b => a * b;
const studentAGrades = [4, 7, 6, 1, 6, 6, 9];
const mult2 = multiply(2);

// ici nous avons une fonction wrapper totalement inutile
const studentAGradesOn20 = studentAGrades.map(grade => {
  return mult2(grade);
});

// la même chose en point free
const studentAGradesOn20PF = studentAGrades.map(mult2);

console.log(studentAGradesOn20); // [8, 14, 12, 2, 12, 12, 18]
```

Point Free Style

Le principe

Attention à la signature : le point free n'est possible que si les signatures coïncident, rappelons nous de l'exemple avec la fonction `parseInt` qui attendait 2 arguments dont la base de conversion (`parseInt(string, radix)`).

On doit donc s'assurer que `parseInt` ne reçoit qu'un argument, on peut utiliser le currying comme on l'a fait précédemment ou utiliser une fonction assez courante **unary**. Son but est simple, elle s'assure que le **seul le premier argument est passé à la fonction**, les autres seront ignorés.

```
const studentAGrades2 = ['4', 7, '6', 1, '6', 6, 9];  
console.log(studentAGrades.map(parseInt));  
  
const unary = (fn) => (arg) => fn(arg);  
  
console.log(studentAGrades.map(unary(parseInt)));
```

La programmation tacite est à utiliser à bon escient. Dans certains cas, où lorsqu'elle est trop poussée, la lisibilité du code peut être altérée. C'est donc un concept à considérer au cas par cas lorsqu'il fait sens.

Si l'on n'applique pas le unary : le point free style va répartir tous les arguments du callback du `map` sur la fonction de traitement `fn`: `[...].map((item, index, array) => fn(item, index, array))`. C'est la raison pour laquelle, le `parseInt` seul ne peut pas fonctionner correctement ici.

Point Free Style Exercices



<https://codesandbox.io/s/exo-point-free-style-0bnlh>



Recursion

Recursion

Le principe

La **récurtivité** n'est pas un concept appartenant à la programmation fonctionnelle, cela dit, elle y est très utilisée. Selon la PF, aucune variable ne doit changer de valeur, ce qui implique qu'on ne devrait pas utiliser les boucles pour cela.

Il est préférable d'utiliser une fonction devrait **s'appeler récursivement** jusqu'à ce qu'une condition soit remplie.

On reconnaît une fonction récursive avec 2 fonctionnalités clé :

- **cas de base** : représente l'ensemble des lignes de code qui permet l'arrêt de la récursion.
- **cas de propagation** : représente l'ensemble des lignes de code où aura lieu la récursivité.

```
const studentAGrades = [4, 7, 6, 1, 6, 6, 9];

function sum(numsArr) {
  let sum = 0;
  numsArr.forEach(nums => {
    sum += nums;
  });

  return sum;
}
```

```
// tant que l'array contient plus d'un élément
// on retourne la somme de l'élément courant au résultat du prochain appel
// avec en argument un nouvel array qui contient tous les éléments de numsArr moins le premier

function recSum(numsArr) {
  if (numsArr.length > 1) { // cas de propagation
    return numsArr[0] + recSum(numsArr.slice(1, numsArr.length));
  }
  // si c'est le dernier élément, on retourne simplement l'élément en question
  return numsArr[0]; // cas de base
}
```


Recursion

La pile d'exécution

C'est la **pile d'exécution** qui garde trace des résultats des appels successifs et qui est en mesure de retourner le résultat final une fois que tous les appels ont été effectués

Cela signifie que **chaque appel doit être conservé en mémoire**, il est donc nécessaire d'être vigilant sur son utilisation.

```
// ici, on ne peut pas suivre l'état manuellement car c'est géré  
// par la pile d'exécution  
// la call stack évolue comme suit  
// 4 + recSum([7, 6, 1, 6, 6, 9]);  
// 4 + 7 + recSum([6, 1, 6, 6, 9]);  
// ...  
// 4 + 7 + 6 + 1 + 6 + 6 + recSum([9]);  
// 4 + 7 + 6 + 1 + 6 + 6 + 9;
```

Step 1

baz()
var x;

Step 2

bar()
var y;
baz()
var x;

Step 3

foo()
var z;
bar()
var y;
baz()
var x;

Recursion

Tail Call Optimisation (TCO)

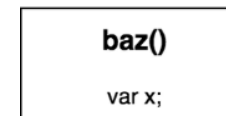
Il s'agit d'une optimisation de la récursion supporté en JS en [strict mode](#).

Lorsque l'appel récursif est la dernière instruction à être évaluée, la pile d'exécution n'a donc pas à garder en mémoire l'état des précédents appels. Dès lors, une fonction récursive qui profite de la TCO jouit d'une **taille de pile fixe** et chaque nouvel appel **n'augmente pas la taille de la pile**.

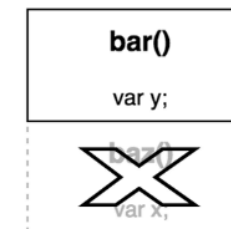
Remarque : le TCO n'est pas implémenté par tous les moteurs Javascript. De plus il faut que la fonction respecte la PTC (proper tail calls), c'est lorsque l'appel à fonction est la dernière chose à être exécutée dans la fonction appelante et que toute valeur est retournée **explicitement**.

```
function recurs() {  
  ...  
  return recurs();  
}
```

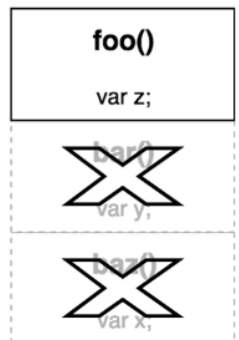
Step 1



Step 2



Step 3



Recursion

Tail Call Optimisation (TCO)

Si on reprend notre exemple avec la somme d'un tableau, la dernière instruction de la fonction retourne bien explicitement l'appel de la fonction elle-même. Il a fallu inverser les conditions pour traiter d'abord le dernier cas et nous avons ajouté un accumulateur en paramètre de la fonction pour gérer nous-même le cumul.

```
function recSum(numsArr) {  
  if (numsArr.length > 1) {  
    return numsArr[0] + recSum(numsArr.slice(1, numsArr.length));  
  }  
  
  return numsArr[0];  
}
```

```
// PTC  
function sumPTC(nums, acc = 0){  
  const sum = nums[0] + acc;  
  if(nums.length === 1) return sum;  
  return sumPTC(nums.slice(1, nums.length), sum);  
}
```

Recursion Exercices



<https://codesandbox.io/s/exo-recursion-8bcoi>



Functor

Functor

Le principe

Définition

Un functor est :

- Un objet
- Qui implémente une méthode map

La méthode map

Elle peut porter un autre nom, mais doit respecter les critères suivants :

- La méthode **map** doit être telle que **$F(f) : F(a) \Rightarrow F(b)$** . (**F pour Functor**)
- La méthode map sert à faire un mapping, à transformer un functor.
- La règle précise juste que si **F** est un type de **functor**, sa méthode map retournera peut-être des données modifiées mais **encapsulées** dans un **functor de même type F**.
- Map ne modifie pas l'instance principale

On ajoute la méthode map
comme propriété

Exemple avec le functor Identity

```
const trace = x => { // on loggue
  console.log('trace',x);
  return x;
};
// Functor
const Identity = value => ({
  map: fn => Identity(fn(value))
});

const u = Identity(2);

// Identity law
u.map(trace); // 2
u.map(x => x).map(trace); // 2
const f = n => n + 1;
const g = n => n * 2;

// Composition law
const r1 = u.map(x => f(g(x)));
const r2 = u.map(g).map(f);
r1.map(trace); // 5
r2.map(trace); // 5
```

le fait de wrapper avec elle-même permet d'effectuer le chainage du map

Functor

Ajouter des propriétés

On peut **enrichir** son functor en ajoutant d'autres propriétés, voici quelques exemples :

value

```
const Functor = (v) => ({
  value: v, // on stocke v dans la propriété value
  map: (fn) => Functor(fn(v)),
});
const square = x => x * x;
const result = Functor(10).map(square);
console.log('Functor simple value', result.value); // 100
```

On peut accéder à la valeur du résultat grâce à value

toString

```
const Identity3 = value => ({
  map: fn => Identity3(fn(value)),
  toString: () => `ma valeur : ${value}`,
});
console.log(`${Identity3(2)} // 'ma valeur : (4)')`);
console.log(Identity3(2).toString()) // 'ma valeur : (4)'
```

Ajout de la propriété toString

log

```
const Identity3 = value => ({
  map: fn => Identity3(fn(value)),
  log: () => console.log(`ma valeur : ${value}`)
});
```

```
Identity3(2).map(square).log();
```

Permet de logger le résultat

Iterable

```
const iterableFunctor = value => ({
  map: fn => iterableFunctor(fn(value)),
  [Symbol.iterator]: function* () {
    for(let i = value; i > 0; i -= 1){
      yield i;
    }
  }
});
const arr = [10, 9, ...iterableFunctor(8)];
console.log('arr', arr); // [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Permet de spread le functor

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/Symbol/iterator

Functor

Quelle utilité ?

S'agissant d'une abstraction, on peut l'utiliser pour implémenter de nombreuses choses utiles de telle manière à ce que cela fonctionne avec n'importe quel type de données.

Dans ces 2 exemples, à l'aide d'un functor (qu'on applique sur un autre Functor), on empêche l'exécution de la chaîne d'opérations si la valeur null ou undefined :

1^{er} functor

```
const Functor = (value) => ({
  value,
  map2: (fn) => Functor(fn(value))
});
```

```
// Create the predicate
const exists = x => (x.value !== undefined && x.value !== null);
```

2^{ème} functor

```
const ifExists = x => ({
  map: fn => ifExists(exists(x) ? x.map2 : null, fn)
});
```

Ici, on rend possible le chainage de map

On applique le 2^{ème} functor sur le 1^{er}

```
ifExists(Functor(null)).map(trace); // nothing happen
ifExists(Functor(undefined)).map(trace); // nothing happen
ifExists(Functor(5)).map(trace).map2(trace); // 5
```

Donc le map qui sera utilisé sera celui du 2^{ème} functor

```
const Functor = (value) => ({
  value,
  map2: (fn) => Functor(fn(value))
});
```

```
// Create the predicate
const exists = x => (x.value !== undefined && x.value !== null);
```

```
const ifExists = x => ({
  map: fn => exists(x) ? x.map2(fn) : x
});
```

Pas de chainage ici

```
ifExists(Functor(null)).map(trace); // nothing happen
ifExists(Functor(undefined)).map(trace); // nothing happen
ifExists(Functor(5)).map(trace).map2(trace); // 5
```

Donc c'est le map2 qui prend le relais

Functor

Exemple : Array

Un exemple de functor bien connu en Javascript est **Array**. Cet [objet possède une méthode map qui permet de modifier les valeurs](#) contenues dans le tableau.

```
const myArray = [1, 2];

const f = x => x * 2;
const g = x => String(x);

// En séquentiel
const transformedMyArray = myArray.map(f).map(g);

// la première loi est respectée car même si les valeurs ont été modifiées, l'identité
est préservée
console.log(myArray, transformedMyArray); // => [1, 2], ["2", "4"]

// Si on teste avec la composition, on doit obtenir le même résultat
const transformedMyArrayWithComposition = myArray.map(item => g(f(item)));
// la seconde loi est respectée
console.log('transformedMyArrayWithComposition',
transformedMyArrayWithComposition); // => ["2", "4"]
```

Array dispose nativement d'une fonction map et respecte bien les lois des functors :

identity law : $\text{functor.map}(x \Rightarrow x) \equiv \text{functor}$

composition law : $\text{functor.map}(x \Rightarrow f(g(x))) \equiv \text{functor.map}(g).map(f)$

Functor

Résumé

Pour résumer, un functor est :

- **un objet**
- **qui implémente la méthode map (peu importe son nom)**
- **Préserve l'identité : $\text{functor.map}(x \Rightarrow x) \equiv \text{functor}$**
- **Préserve la composition : $\text{functor.map}(x \Rightarrow f(g(x))) \equiv \text{functor.map}(g).map(f)$**

Functor Exercices



<https://codesandbox.io/s/exo-functor-tyfqb>





MERCI