PG104 - Programmation Fonctionnelle

D. Renault

Enseirb-Matmeca

10 février 2025, v.1.2.5

https://www.labri.fr/perso/renault/working/teaching/functional/functional.php

Citoyenneté de 1ère classe

Principe général

Écrire du code profitant du caractère hautement composable des fonctions.

- Les expressions mathématiques sont une premier signe de l'expressivité que l'on peut obtenir en composant des fonctions.
- Naturellement, on attend plus d'un langage de programmation.
- D'où la question de lister les qualités que l'on peut attendre des fonctions au sein des programmes $\Rightarrow$ **citoyenneté de 1ère classe**.

Définition (1ère classe)

Une construction d'un langage de programmation est dite **citoyenne de 1ère classe** (*1st class citizen*) si le langage l'autorise à :

- [**Nommage**] être nommée et affectée dans une variable ;
- [**Création**] être créée à la demande dans n'importe quel contexte ;
- [**Argument**] être passée comme argument à une fonction ;
- [**Retour**] être retournée comme résultat d'une fonction ;
- [**Stockage**] apparaître dans n'importe quelle structure de données.

- La 1ère classe n'est pas une fin en soi : certaines techniques sont appliquables même si toutes les propriétés ne sont pas vérifiées.
- Chaque propriété peut être vue comme une direction à explorer.
- La notion de 1ère classe peut s'appliquer à d'autres constructions que les fonctions : les macros, les modules, les classes ...

1ère ou 2ème classe ?

Exemple

En Ecmascript, les chaînes de caractères sont de 1ère classe :

```
let aString = "hello";                  // can be named, created and assigned
function repeat(str) {                  // can be passed to a function
  return str + str;  }                  // and returned from it
repeat(aString);                        //
let anArr = [ aString; "world" ];       // can be stored in a data structure
```

Contre-exemples

- En C, les fonctions **ne sont pas** de 1ère classe : on ne peut pas les créer à la demande.
- En C++, les objets sont de 1ère classe, mais les classes elle-mêmes **ne le sont pas**.

Les fonctions de 1ère classe

En Ecmascript, les fonctions **sont** des valeurs de 1ère classe :

- **Nommage** : être nommée et affectée dans une variable ;

```
let aFun = function (x) { return x + 1; };
```

Les fonctions de 1ère classe

En Ecmascript, les fonctions **sont** des valeurs de 1ère classe :

- **Nommage** : être nommée et affectée dans une variable ;

```
let aFun = function (x) { return x + 1; };
```

- **Création** : être créée à la demande dans n'importe quel contexte ;

```
(function (x) { return x + 1; }) + 'is fun'
```

Les fonctions de 1ère classe

En Ecmascript, les fonctions **sont** des valeurs de 1ère classe :

- **Nommage** : être nommée et affectée dans une variable ;

```
let aFun = function (x) { return x + 1; };
```

- **Création** : être créée à la demande dans n'importe quel contexte ;

```
(function (x) { return x + 1; }) + 'is fun'
```

- **Argument** : être passée comme argument à une fonction ;

```
console.log(function (x) { return x + 1; });
```

Les fonctions de 1ère classe

En Ecmascript, les fonctions **sont** des valeurs de 1ère classe :

- **Nommage** : être nommée et affectée dans une variable ;

```
let aFun = function (x) { return x + 1; };
```

- **Création** : être créée à la demande dans n'importe quel contexte ;

```
(function (x) { return x + 1; }) + 'is fun'
```

- **Argument** : être passée comme argument à une fonction ;

```
console.log(function (x) { return x + 1; });
```

- **Retour** : être retournée comme résultat d'une fonction ;

```
function addOne(f) { return ((x) ⇒ f(x+1)); }
```

Les fonctions de 1ère classe

En Ecmascript, les fonctions **sont** des valeurs de 1ère classe :

- **Nommage** : être nommée et affectée dans une variable ;

```
let aFun = function (x) { return x + 1; };
```

- **Création** : être créée à la demande dans n'importe quel contexte ;

```
(function (x) { return x + 1; }) + 'is fun'
```

- **Argument** : être passée comme argument à une fonction ;

```
console.log(function (x) { return x + 1; });
```

- **Retour** : être retournée comme résultat d'une fonction ;

```
function addOne(f) { return ((x) ⇒ f(x+1)); }
```

- **Stockage** : apparaître dans n'importe quelle structure de données.

```
let anArray = [ Math.sin, (x) ⇒ x + 1 ];
```

Dans la suite, nous explorons chacune des propriétés de la 1ère classe :

- **Argument** : prendre en paramètre des fonctions ;
- **Retour** : être retourné comme résultat d'une fonction ;
- **Stockage** : apparaître dans une structure de données ;
- **Composition** : mélanger les précédentes propriétés.

1ère classe – Prendre en paramètre des fonctions

Dans quel but ?

- Une fonction est à considérer comme un morceau de code portable.
- Prendre une fonction en paramètre est donc largement plus flexible que prendre un entier ou une chaîne de caractères.

Définition (Fonction d'ordre supérieur)

Une **fonction d'ordre supérieur** (*higher-order function*) est une fonction dont au moins l'un des paramètres est une fonction.

Nous avons déjà rencontré de telles fonctions avec `map`, `filter` et `reduce`.
Quels besoins font apparaître naturellement ces fonctions dans le code ?

Exemple de fonction d'ordre supérieur

La fonction `loop` est un exemple de fonction d'ordre supérieur qui répète un certain nombre de fois le `block` passé en paramètre :

```
function loop(block, init, times) { // block is a 1-parameter function
  let res = init;
  for (let i = 0; i < times; i++) {
    res = block(res);
  }
  return res;
}
```

Comment permettre de paramétrer le moment où la boucle s'arrête ?

Exemple de fonction d'ordre supérieur

La fonction `loop` est un exemple de fonction d'ordre supérieur qui répète un certain nombre de fois le `block` passé en paramètre :

```
function loop(block, init, times) { // block is a 1-parameter function
  let res = init;
  for (let i = 0; i < times; i++) {
    res = block(res);
  }
  return res;
}
```

Comment permettre de paramétrer le moment où la boucle s'arrête ?

⇒ En ajoutant un paramètre fonctionnel `until` renvoyant un booléen.

Exemple de fonction d'ordre supérieur

La fonction `loop` est un exemple de fonction d'ordre supérieur qui répète un certain nombre de fois le `block` passé en paramètre :

```
function loopUntil(block, init, until) {   // block is a 1-parameter function
  let res = init;                          // until is a 1-parameter function
  for (let i = 0; until(i); i++) {
    res = block(res);
  }
  return res;
}
```

La fonction `loop` peut s'écrire à partir de `loopUntil` comme suit :

```
let loop = (block, init, times) ⇒ loopUntil(block, init, (i) ⇒ i < times);
```

Définition (Généralisation)

Une **généralisation** fonctionnelle (*func. generalization*) d'une fonction f consiste à remplacer une partie du corps de f par un appel à une fonction qui est ajoutée aux paramètres de f.

- Les généralisations rendent le code plus adaptable et réutilisable.
- Elles se composent bien avec les fonctions anonymes qui peuvent fournir les paramètres fonctionnels supplémentaires.

Exemple de généralisation : countChars

Reprenons l'exemple de la fonction comptant les occurrences d'un caractère dans une chaîne :

```
function countChars(str, c) {
    let res = 0;
    for (let i = 0; i < str.length; i++) {
        if (str[i] === c)
          res += 1;
    }
    return res;
}
```

Comment peut-on faire pour que `countChars` compte les caractères dans un ensemble donné (par exemple toutes les lettres de l'alphabet) ?

Exemple de généralisation : countChars

Reprenons l'exemple de la fonction comptant les occurrences d'un caractère dans une chaîne :

```
function countMatches(str, matchFun) { // matchFun is a 1-parameter function
    let res = 0;
    for (let i = 0; i < str.length; i++) {
        if (matchFun(str[i]))
          res += 1;
    }
    return res;
}
```

La fonction `countChars` peut s'écrire à partir de `countMatches` comme suit :

```
let countChars = (str, c) ⇒ countMatches(str, (s) ⇒ s === c);
```

Exemple de généralisation : sortThree

Prenons l'exemple d'une fonction triant un tableau de 3 éléments :

```
function sortThree([a, b, c]) {
    if (a < b) {  // a < b
        return (c < a) ? [c, a, b] :
               (c < b) ? [a, c, b] :
                         [a, b, c];
    } else {      // b <= a
        return (c < b) ? [c, b, a] :
               (c < a) ? [b, c, a] :
                         [b, a, c];
}}
```

Comment peut-on généraliser cette fonction ?

Exemple de généralisation : sortThree

Prenons l'exemple d'une fonction triant un tableau de 3 éléments :

```
function sortThreeGen([a, b, c], cmpFun) { // cmpFun is a 2-param function
    if (cmpFun(a, b)) { // a < b
        return (cmpFun(c, a)) ? [c, a, b] :
               (cmpFun(c, b)) ? [a, c, b] :
                                [a, b, c];
    } else {              // b <= a
        return (cmpFun(c, b)) ? [c, b, a] :
               (cmpFun(c, a)) ? [b, c, a] :
                                [b, a, c];
}}
```

La fonction `sortThree` peut s'écrire à partir de `sortThreeGen` comme suit :

```
let sortThree = (arr) ⇒ sortThreeGen(arr, (a, b) ⇒ a < b);
```

Exemple de généralisation : factorisation

Les généralisations peuvent servir à factoriser du code, par exemple :

```
let numTests = 0;
let numPassed = 0;

function testEmptyIsEmpty() {
    const aSet = setEmpty();
    const res = setIsEmpty(aSet);
    numTests++;
    if (res) numPassed++;
    return res;
}
function testNonEmptyIsNotEmpty() {
    const aSet = setEmpty();
    setAdd(aSet, 1);
    const res = !setIsEmpty(aSet);
    numTests++;
    if (res) numPassed++;
    return res;
}
```

Exemple de généralisation : factorisation

Les généralisations peuvent servir à factoriser du code, par exemple :

```
let numTests = 0;
let numPassed = 0;

function testEmptyIsEmpty() {
    const aSet = setEmpty();
    const res = setIsEmpty(aSet);
    numTests++;
    if (res) numPassed++;
    return res;
}
function testNonEmptyIsNotEmpty() {
    const aSet = setEmpty();
    setAdd(aSet, 1);
    const res = !setIsEmpty(aSet);
    numTests++;
    if (res) numPassed++;
    return res;
}
```

Exemple de généralisation : factorisation

Les généralisations peuvent servir à factoriser du code, par exemple :

```
let numTests = 0;
let numPassed = 0;

function testEmptyIsEmpty() {
    const aSet = setEmpty();
    const res = setIsEmpty(aSet);
    numTests++;
    if (res) numPassed++;
    return res;
}
function testNonEmptyIsNotEmpty() {
    const aSet = setEmpty();
    setAdd(aSet, 1);
    const res = !setIsEmpty(aSet);
    numTests++;
    if (res) numPassed++;
    return res;
}
```

```
let numTests = 0;
let numPassed = 0;

function testEmptyIsEmpty() {
    const aSet = setEmpty();
    return setIsEmpty(aSet);
}
function testNonEmptyIsNotEmpty() {
    const aSet = setEmpty();
    setAdd(aSet, 1);
    return !setIsEmpty(aSet);
}
function runTest(aTest) {
    const res = aTest();
    numTests++;
    if (res) numPassed++;
    return res;
}
```

Exemple de généralisation : stratégie

Les généralisations peuvent servir à définir des **stratégies** :

```
function playHeadsOrTails() {
    let gain = 10;
    while ((gain > 0) && (gain < 20)) {
        let bet = gain;
        gain += (Math.random() > 0.5) ? bet : -bet;
    }
    return gain;
}
```

Comment peut-on généraliser cette fonction ?

Exemple de généralisation : stratégie

Les généralisations peuvent servir à définir des **stratégies** :

```
function playHeadsOrTailsGen(betFun, contFun) { // betFun and contFun
    let gain = 10;                               // are 1-parameter functions
    while (contFun(gain)) {
        let bet = betFun(gain);
        gain += (Math.random() > 0.5) ? bet : -bet;
    }
    return gain;
}
```

La fonction `playHeadsOrTails` peut s'écrire à partir de l'autre comme suit :

```
let playHeadsOrTails = () => playHeadsOrTailsGen((a) => a , (a) => a <20);
```

Généralisations et généricité

Définition (Généricité – informelle)

Une fonction est dite **générique** si elle peut être utilisée correctement sur différents *types* de valeurs.

Exemple

La dernière fonction `countMatches` est en fait une fonction générique :

```
countMatches("abcde",      (c) ⇒ c>="a" && c<="e"); // → 5
countMatches([1,2,3,4,5], (c) ⇒ c%2 === 0);        // → 2
```

- Les généralisations entraînent de la généricité.
- Cette généricité est une forme particulière de **polymorphisme**.

```
function countMatches<T>(arr: T[], match: (T) ⇒ boolean) : number
```

Caveat sur les généralisations :

- Il y a un compromis entre ajouter des paramètres fonctionnels et garder des prototypes de fonctions raisonnables.
- Nous verrons plus tard d'autres techniques pour transporter plusieurs paramètres fonctionnels en même temps.

1ère classe – Renvoyer des fonctions

Dans quel but ?

- Une fonction renvoyant une fonction, c'est un **constructeur** de fonctions.
- Si les fonctions sont considérées comme des valeurs centrales, avoir des facilités pour les construire est un bonus.
- Mieux, donner des paramètres à ces constructeurs permet d'adapter les fonctions ainsi construites.

Nous avons déjà rencontré de tels exemples dans les slides précédents.
Tâchons de mettre des noms sur les techniques associées.

Spécialisation (1/2)

Définition (Spécialisation)

Une **spécialisation** fonctionnelle (*specialization*) d'une fonction f sur un de ses paramètres p consiste à retirer p des paramètres de f et le remplacer dans le corps de f par une valeur donnée.

Exemple

```
function add(x, y) { return x + y; }
function add0ne(y) { return add(1, y);} // Specialization of add on x = 1
```

- Il est aussi possible d'utiliser le terme d'**application partielle**.

Spécialisation (2/2)

- Une spécialisation peut être réalisée à l'aide de **paramètres par défaut** :

```
function add(x, y = 1) { return x + y; }
add(5);          // → 6
```

- Une spécialisation peut être réalisée à l'aide de **fonctions anonymes** :

```
const addOne = function (x) { return add(x, 1); }
addOne(5);       // → 6
```

(e.g pour les langages ne permettant pas les paramètres par défaut)

- Mieux, il est possible de **paramétrer** la spécialisation :

```
const addAny = (v) ⇒ function (x) { return add(x, v); }
addAny(660)(6);  // → 666,        addAny(660) is the function ''adding 660''
```

Curryfication (1/3)

Définition (Forme curryfiée)

La **forme curryfiée** d'une fonction (*curried form*) est une fonction réalisant le même calcul, prenant ses paramètres un par un dans des fonctions successives :

$(x_1,\ x_2,\ \ldots,\ x_n) \Rightarrow \{\ \text{body}\ \} \quad \rightarrow \quad (x_1) \Rightarrow (x_2) \Rightarrow \ldots \Rightarrow (x_n) \Rightarrow \{\ \text{body}\ \}$

Exemple

```
function add(x, y) { return x + y; }                    // uncurried form

function addCurrified1(x) {                             // curried form (function)
    return function (y) { return x + y; }               //
}                                                       //

const addCurrified2 = (x) => (y) => { return x + y; }; // curried form (anonymous)
```

Curryfication (2/3)

- Tant qu'elle n'a pas reçu tous ses paramètres, l'appel à une fonction curryfiée renvoie une fonction :

```
function log = (lvl) ⇒ (date) ⇒ (msg) ⇒ `${lvl} (${date}): ${msg}`;
log('DEBUG');                                  // → Function
log('DEBUG')('12:34');                         // → Function
log('DEBUG')('12:34')('The server is on fire'); // → String
```

Ce sont des exemples de fonctions renvoyant des fonctions.

- La curryfication est une forme de **contrôle de l'évaluation** : le calcul n'est effectué qu'une fois le dernier argument fourni.
- Les fonctions curryfiées sont facilement spécialisables :

```
const debug = log('DEBUG');                // Specialize on first parameter
debug('13:56')('Out of water error');      // Reuse the specialized function
```

Curryfication (3/3)

Il est possible de curryfier automatiquement une fonction en Ecmascript.
(mais cela ne fait pas partie de la bibliothèque standard)

Extrait de la documentation de la fonction `curry` de la bibliothèque `lodash` :

```
import _ from 'lodash';
```

```
const abc = function(a, b, c) { return [a, b, c]; };

const curried = _.curry(abc);   // transform into curried form

typeof curried(1); // "function"
curried(1)(2)(3);  // ⇒ [1, 2, 3]
curried(1, 2)(3);  // ⇒ [1, 2, 3]
curried(1, 2, 3);  // ⇒ [1, 2, 3]
```

Dans certains langages de programmation (OCaml, Haskell, ...), les fonctions sont automatiquement curryfiées.

Généralisations et spécialisations

Les généralisations et les spécialisations se complètent naturellement :

A generalization

```
function countMatches(coll, matchFun) { // matchFun is a 1-parameter function
    let res = 0;
    for (let i = 0; i < coll.length; i++) {
        if (matchFun(coll[i]))
          res += 1;
    }
    return res;
}
```

Some specializations

```
// Tells if a string contains dangerous HTML characters such as brackets
const hasBrackets  = (coll) ⇒ countMatches(coll, (c) ⇒ /[<>]/.test(c)) > 0;
// Tells if an array contains undefined elements
const hasUndefined = (coll) ⇒ countMatches(coll, (c) ⇒ c === undefined) > 0;
```

Exemple de générateurs : pluck

La fonction `pluck` permet de créer aisément des accesseurs dans des objets :

```
function pluck(aField) {                    // Example of a curried function
    return function(obj) {                  // taken from [Fogus]
        return (obj && obj[aField]);
    };}
```

Exemple de générateurs : pluck

La fonction `pluck` permet de créer aisément des accesseurs dans des objets :

```
function pluck(aField) {                    // Example of a curried function
    return function(obj) {                  // taken from [Fogus]
        return (obj && obj[aField]);
    };}
```

Quelques manipulations d'une bibliothèque de livres :

```
const books = [
    { id: 7, title: 'Ender\'s game', author: 'Scott Card Orson' },
    { id: 1, title: 'Wyrd sisters',  author: 'Pratchett Terry' },
    { id: 9, title: 'Elevation',     author: 'Brin David'  },          ];
```

```
const getTitle  = pluck('title');
books.map(getTitle); // → [ "Ender's game", 'Wyrd sisters', 'Elevation' ]
const getAuthor  = pluck('author');
books.filter((b) ⇒ getAuthor(b).startsWith('P'));
                    // → [ { title: 'Wyrd sisters', author: 'Pratchett Terry' } ]
```

Exemple de générateurs : compareWith

La fonction `compareWith` construit des comparateurs utilisables avec `sort` :

```
function compareWith(aField) {
    return function(obj1, obj2) {
        const v1 = pluck(aField)(obj1), v2 = pluck(aField)(obj2);
        return (v1 < v2) ? -1 : (v1 > v2) ? 1 : 0;
    };}
```

Exemple de générateurs : compareWith

La fonction `compareWith` construit des comparateurs utilisables avec `sort` :

```
function compareWith(aField) {
    return function(obj1, obj2) {
        const v1 = pluck(aField)(obj1), v2 = pluck(aField)(obj2);
        return (v1 < v2) ? -1 : (v1 > v2) ? 1 : 0;
    };}
```

Quelques exemples de tris avec la bibliothèque précédente :

```
function sortArray(arr, cmp) {                  // return a sorted copy of an array
    const arrC = [...arr];
    arrC.sort(cmp);
    return arrC;
}
sortArray(books, compareWith('author'));    // books sorted with ids [9, 1, 7]
sortArray(books, compareWith('title'));     // books sorted with ids [9, 7, 1]
```

Les comparateurs peuvent même être étendus de manières variées :

- Pour inverser un comparateur :

```
function compareWithReverse(field) {
    return function(obj1, obj2) {
        return - compareWith(field)(obj1, obj2);
    };}
sortArray(books, compareWithReverse('title'));
```

- Pour composer plusieurs comparateurs :

```
function compareWithFields(...fields) {
    return function(obj1, obj2) {
        return fields.reduce((b, aField) => {
            return (b !== 0) ? b : compareWith(aField)(obj1, obj2);
        }, 0);
    };}
sortArray(books, compareWithFields('id', 'title', 'author'));
```

Exemple : décorateurs

Définition (Décorateur)

Un **décorateur** fonctionnel prend une fonction f et renvoie une fonction avec le même prototype, avec un comportement augmenté.

Par exemple, pour logguer les appels à une fonction :

```
function log(aFun, aMsg) {
    return function(...args) {
        console.log(`${aMsg} : ${aFun.name}(${args})`);
        const res = aFun(...args);                    // Call aFun
        console.log(`${aMsg} returns ${res}`);
        return res;
}} // Beware when using with recursive functions
```

```
function fact(n) {return (n<=1) ? 1 : n*fact(n-1); }
fact = log(fact, "Fact");          // Replace 'fact'
fact(5);                  // Calls are properly logged
```

```
Fact : fact(3), Fact : fact(2)
Fact : fact(1), Fact returns 1
Fact returns 2, Fact returns 6
```

À comparer à la fonction `wrap` des bibliothèques Underscore et Lodash.

Conclusion (partielle) sur la 1ère classe

- La 1ère classe est une propriété listant un ensemble de modes d'utilisation des objets, ici les fonctions.
- A chacun de ces modes d'utilisations correspondent des techniques de programmation.
  - Les **généralisations** utilisent des paramètres fonctionnels pour écrire du code générique, réutilisable.
  - Les **spécialisations** renvoient des fonctions dans lesquelles on a fixé les valeurs de certains paramètres.
- Généralisations et spécialisations fonctionnent seules, mais se combinent avec profit, les secondes aidant à la réutilisation des premières.

Quelques lectures . . .


Fogus, M.: Functional Javascript.
O'Reilly, 2013.


Narbel, P.: Programmation fonctionnelle, générique et objet : Une introduction avec le langage OCaml.
Vuibert, 2005.


Reade, C.: Elements of Functional Programming.
Addison-Wesley, 1989.


Bird, R. et P. Wadler: Introduction to Functional Programming.
Prentice Hall, 1988.


Okasaki, C.: Purely functional data structures.
Cambridge University Press, 1999.


Narbel, P.: Techniques Avancées de Programmation.
Cours de Master 2 à l'Université de Bordeaux.