PG104 - Programmation Fonctionnelle

D. Renault

Enseirb-Matmeca

17 février 2025, v.1.2.5

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

1ère classe – Stocker des fonctions

Dans quel but ?

- Plusieurs fonctions réunies ensemble dans une même structure peuvent facilement se **composer** ou plus généralement **partager** des liens ;

  Comment tirer profit des liens ainsi créés ?

- Ranger des fonctions dans des structures de données, c'est considérer les fonctions comme des données **dynamiques** :

  transportables, partageables, réutilisables . . .

  Quels genre de données profitent de composants fonctionnels ?

Exemple : codage et décodage

Reprenons l'exemple de fonctions de chiffrement et déchiffrement :

```
function makeCypher(key) {
    function cypher (str, offset) { ... }
    function encode(str) { return cypher(str, key);  }
    function decode(str) { return cypher(str, -key); }
    return { encode: encode, decode: decode };
}

const c = makeCypher(42);
c.decode(c.encode('Hello')); // → 'Hello'
```

- Les deux fonctions `encode` et `decode` sont inverses l'une de l'autre.
- Il est sensé de les garder ensemble et proches dans un dictionnaire.

Exemple : objets-enregistrements (1/2)

Les **enregistrements** (`record`, `struct` ...) permettent aisément de regrouper ensemble des données et des traitements (ici des nombres complexes) :

```
function makeComplex(re, im) {









    return { re: re,       im: im };

}
```

```
let c1 = makeComplex(1,0);
let cI = makeComplex(0,1);

c1;
// → { re: 1, im: 0 }

cI;
// → { re: 0, im: 1 }
```

Exemple : objets-enregistrements (1/2)

Les **enregistrements** (`record`, `struct` ...) permettent aisément de regrouper ensemble des données et des traitements (ici des nombres complexes) :

```
function makeComplex(re, im) {




    return { re: re,       im: im };

}
```

```
let c1 = makeComplex(1,0);
let cI = makeComplex(0,1);

c1.add(cI);
// → Error

cI.mul(cI);
// → Error
```

Exemple : objets-enregistrements (1/2)

Les **enregistrements** (`record`, `struct` ...) permettent aisément de regrouper ensemble des données et des traitements (ici des nombres complexes) :

```
function makeComplex(re, im) {
    function add(c) {



    }




    return { re: re,        im: im,
             add: add };
}
```

```
let c1 = makeComplex(1,0);
let cI = makeComplex(0,1);

c1.add(cI);
// → undefined

cI.mul(cI);
// → Error
```

Exemple : objets-enregistrements (1/2)

Les **enregistrements** (`record`, `struct` ...) permettent aisément de regrouper ensemble des données et des traitements (ici des nombres complexes) :

```
function makeComplex(re, im) {
    function add(c) {
        return makeComplex(c.re + re,
                           c.im + im);
    }

    return { re: re,       im: im,
             add: add };
}
```

```
let c1 = makeComplex(1,0);
let cI = makeComplex(0,1);

c1.add(cI);
// → { re: 1, im: 1, add.. }

cI.mul(cI);
// → Error
```

Exemple : objets-enregistrements (1/2)

Les **enregistrements** (`record`, `struct` ...) permettent aisément de regrouper ensemble des données et des traitements (ici des nombres complexes) :

```
function makeComplex(re, im) {
    function add(c) {
        return makeComplex(c.re + re,
                           c.im + im);
    }
    function mul(c) {
        return makeComplex(c.re*re - c.im*im,
                           c.re*im + c.im*re);
    }
    return { re: re,       im: im,
             add: add,     mul: mul };
}
```

```
let c1 = makeComplex(1,0);
let cI = makeComplex(0,1);

c1.add(cI);
// → { re: 1, im: 1, add, mul.. }

cI.mul(cI);
// → { re: -1, im: 0, add, mul.. }
```

Exemple : objets-enregistrements (2/2)

Les **enregistrements** permettent aisément de regrouper ensemble des données et des traitements (ici des données génériques dans un ensemble) :

```
struct generic_t {
   int (*cmp)(const void *, const void *); // a function for comparing values
   void* (*copy)(const void *);            // a function to copy values
   void (*del)(void *);                    // a function to free values
};

struct set* set__empty(const struct generic_t*);   // allocates a generic set
```

- Le résultat est une structure de données contenant des fonctions.
- Il est ainsi possible de créer des valeurs ressemblant à des **objets** (sans la complexité du monde objet et l'accès au `this`).

Exemple : modules

Définition (Module)

Un **module** est un espace de nommage avec des possibilités de masquer ou non les éléments à l'intérieur.

Le pattern 'Module' en Ecmascript

```
const aModule = ( function () {
    const conf = { caching : true,
                   language : 'en' };
    return {
        reportConfig :  function () {
            console.log(`Caching ${conf.caching}`); },
        updateCaching : function (caching) {
            conf.caching = caching; }
    }
} ) ();
aModule.updateCaching(false);
aModule.reportConfig(); // Log : 'Caching false'
```

Exemple : modules

Définition (Module)

Un **module** est un espace de nommage avec des possibilités de masquer ou non les éléments à l'intérieur.

Le pattern 'Module' en Ecmascript

```
const aModule = ( function () {
    const conf = { caching : true,
                   language : 'en' };
    return {
        reportConfig :  function () {
            console.log(`Caching ${conf.caching}`); },
        updateCaching : function (caching) {
            conf.caching = caching; }
    }
} ) ();
aModule.updateCaching(false);
aModule.reportConfig(); // Log : 'Caching false'
```

Exemple : modules

Définition (Module)

Un **module** est un espace de nommage avec des possibilités de masquer ou non les éléments à l'intérieur.

Le pattern 'Module' en Ecmascript

```
const aModule = ( function () {
    const conf = { caching : true,
                   language : 'en' };
    return {
        reportConfig :  function () {
            console.log(`Caching ${conf.caching}`); },
        updateCaching : function (caching) {
            conf.caching = caching; }
    }
} ) ();
aModule.updateCaching(false);
aModule.reportConfig(); // Log : 'Caching false'
```

Private

Public

Caveat sur le stockage de fonctions

Les constructions ainsi évoquées sont souvent des **simplifications**.
Il existe des constructions plus complexes à un tel niveau de granularité :

- la couche **objet** d'Ecmascript est plus riche que ce que peuvent exprimer les enregistrements (prototypes, accès au `this` ...)
- la couche **modulaire** d'Ecmascript a évolué et apporte plus de fonctionnalités (gestion des dépendances, chargement asynchrone ...)

Néanmoins, elles permettent de faire le lien entre les différents styles de programmation.

Et elles démontrent l'intérêt de stocker ensemble des fonctions.

Callbacks (1/2)

Les fonctions peuvent aussi être utilisées par et pour elle-mêmes.

Définition (Callback)

Un **callback** est une fonction destinée à être exécutée lorsqu'un événement particulier se produit au cours de l'exécution.

- Les navigateurs Web représentent les pages HTML sous la forme d'un arbre suivant une spécification nommée **Document Object Model** ;
- Cette spécification définit en particulier un ensemble d'événements auxquels attacher des callbacks :

  `click`, `drag`, `drop`, `focus`, `input`, `keypress`, `load` ...

- Le langage permet alors d'associer événements et callbacks :

```
const button = document.querySelector('button');
button.addEventListener('click',
    event ⇒ { button.innerHTML = `#Clicks : ${event.detail}`; });
```

Callbacks (2/2)

Les utilisations des callbacks ne se limitent pas aux interfaces graphiques.

- De nombreuses fonctions de l'API de Node.js utilisent des callbacks pour gérer les erreurs (*error callback convention*).
- En particulier, les opérations fonctionnant de manière asynchrone, comme celles faisant appel au système de fichiers :

```
const aPath = '/some/doomed/file';          // fs.unlink deletes a file
fs.unlink(aPath, (err) ⇒ {                  // callback called on completion
  if (err) throw err;                       // re-throw the error if ever
  console.log(`successfully deleted '${aPath}'`); // otherwise just log
});
```

Un paramètre fonctionnel ajouté à une fonction et censé être appelé à sa terminaison s'appelle une **continuation**.

Deux mots sur les continuations

Pour simplifier, une **continuation** est un callback appelé après le retour de la fonction sur son résultat. Par exemple :

Version classique

```
function fibo(n) {
  if (n <= 1) {
    return 1;
  } else {
    return fibo(n-1) + fibo(n-2);
  }
}



[3,4,5,6].map(fibo); // → [3,5,8,13]
```

Version avec continuations

```
function fiboCps(n, cont) {
  if (n <= 1) {
    return cont(1);
  } else {
    return fiboCps(n-2, (a) ⇒
           fiboCps(n-1, (b) ⇒
                        cont(a+b)
           ));
} }

const myFibo = (e) ⇒ fiboCps(e, (r) ⇒ r);
[3,4,5,6].map(myFibo); // → [3,5,8,13]
```

Ce style d'écriture de code est appelé **continuation-passing style** ou CPS.

Représentation des données par les fonctions (1/3)

Les fonctions peuvent servir pour représenter des données par elle-mêmes.

- prédicats,
- comparateurs,
- stratégies,
- callbacks,
- ...

En tant que données, les fonctions brillent surtout par leur **expressivité**.

Représentation des données par les fonctions (2/3)

Idée

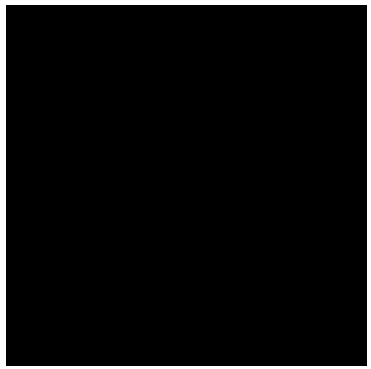
User de **fonctions caractéristiques** pour décrire des ensembles complexes.

Étant donné un ensemble E, sa fonction caractéristique est la fonction χ_E :

$$\chi_E(t) = \begin{cases} 1 & \text{si } t \in E \\ 0 & \text{sinon} \end{cases}$$

- Ces fonctions permettent de représenter des ensembles de **données** de manière générique.
- Le type des données et la forme des ensembles sont très versatiles :

```
function setFull(p) {
  return true;
}
```



Représentation des données par les fonctions (2/3)

Idée

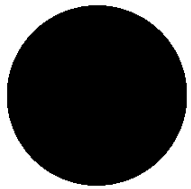
User de **fonctions caractéristiques** pour décrire des ensembles complexes.

Étant donné un ensemble E, sa fonction caractéristique est la fonction χ_E :

$$\chi_E(t) = \begin{cases} 1 & \text{si } t \in E \\ 0 & \text{sinon} \end{cases}$$

- Ces fonctions permettent de représenter des ensembles de **données** de manière générique.
- Le type des données et la forme des ensembles sont très versatiles :

```
function setCircle(p) {
  return ((p.x-1)**2 + (p.y-1)**2) <= 0.25;
}
```



Représentation des données par les fonctions (2/3)

Idée

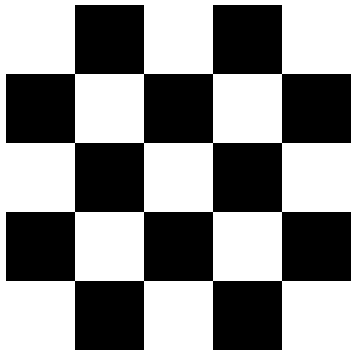
User de **fonctions caractéristiques** pour décrire des ensembles complexes.

Étant donné un ensemble E, sa fonction caractéristique est la fonction χ_E :

$$\chi_E(t) = \begin{cases} 1 & \text{si } t \in E \\ 0 & \text{sinon} \end{cases}$$

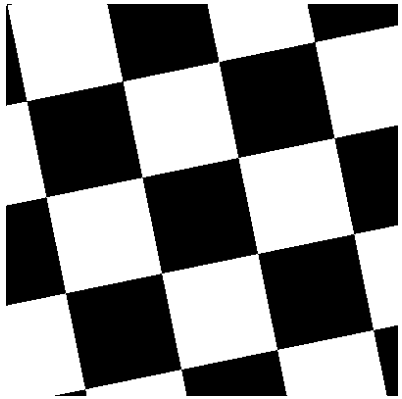
- Ces fonctions permettent de représenter des ensembles de **données** de manière générique.
- Le type des données et la forme des ensembles sont très versatiles :

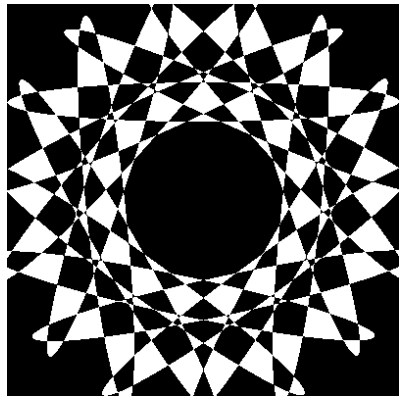
```
function setGrid(p) {
  let rx = Math.floor(x*2.5);
  let ry = Math.floor(y*2.5);
  return (rx+ry) % 2 == 1;
}
```



Représentation des données par les fonctions (3/3)

Ces fonctions peuvent ensuite être transformées en d'autres fonctions.
Quelques exemples pour les ensembles de points :





Idée

Représenter les données par des fonctions offrir des possibilités étendues par rapport aux valeurs classiques.

- Générateur de listes de lectures sur critères de préférence ;
- Familles de stratégies (MinMax, ANN ...) pour les jeux ...

Exemple : validateurs [Fogus]

Idée

Créer des **validateurs** vérifiant si les valeurs ont des propriétés adéquates.

```
function validator(message, fun) {
  const f = function(...args) { return fun(...args); };
  f['message'] = message;
  return f;
}

function checker(...validators) {
  return function(obj) {
    return _.reduce(validators,
                    function(errs, aValidator) {
      return (aValidator(obj)) ?
                errs : [ ...errs, aValidator.message ];
    }, []);
  };
}
```

```
const _ = require("underscore");

const argNumber = validator(
    "arg should be a number", _.isNumber);
const argPos = validator(
    "arg should be > 0", (n) => n > 0);
const argEven = validator(
    "arg should be even", (n) => n % 2 === 0);

// A checker that combines 2 validators
const chk1 = checker(argNumber, argPos);

// Examples
chk1(5);     // → []
chk1(-3);    // → ['arg should be > 0']
chk1("lol"); // → ['arg should be a number',
             //    'arg should be > 0']
```

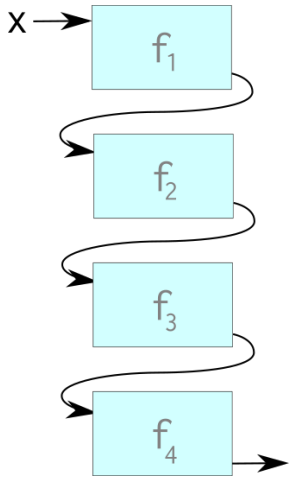
Profite ici de la facilité de composer des fonctions simples (les validateurs) en des fonctions complexes (le checker).

Composer des fonctions

Dans quel but ?

- Composer des fonctions entre elles peut sembler une évidence, mais on a souvent tendance à penser les fonctions de manière indépendante.

  Quelles techniques permettent de profiter de cette capacité de composition ?



- Accessoirement, comment concevoir des bibliothèques de fonctions qui facilitent ces compositions ?

Exemple : pipeline

Composer les fonctions de manière à ce que chacune envoie son résultat en paramètre à la suivante peut se faire facilement avec un **pipeline** :

```
import * as _ from 'underscore';
```

```
function pipeline(seed, ...args) {
    return _.reduce(args,
        function(l,r) { return r(l); },
    seed);
};

pipeline("abba est un groupe suedois",
         (s) ⇒ s.split(" "),          // → ["abba", "est", "un", ... ]
         (a) ⇒ a.map(capitalize),     // → ["Abba", "Est", "Un", ... ]
         (a) ⇒ a.join(' '));          // → "Abba Est Un Groupe Suedois"
```

Principe d'un pipeline

Une donnée qui est transformée au fur et à mesure d'un calcul.

Exemple : chain avec lodash

Plusieurs bibliothèques proposent de composer les fonctions en pipeline.
Extrait de la documentation de la fonction `chain` de la bibliothèque `lodash` :

```
import * as _ from 'lodash-es';
```

```
const users = [ { 'user': 'barney',  'age': 36 },
                { 'user': 'fred',    'age': 40 },
                { 'user': 'pebbles', 'age': 1  } ];

const youngest = _
  .chain(users)
  .sortBy('age')
  .map((o) => `${o.user} is ${o.age}`)
  .head()
  .value();                                  // => 'pebbles is 1'
```

- Le chaînage est effectué ici en chaînant les appels de méthodes.
- La fonction `chain` débute la chaîne et la méthode `value` la termine.

Exemple : ifElse avec Ramda

La composition peut même se permettre de ne pas être linéaire.
Exemple d'utilisation de la fonction `ifElse` de la bibliothèque `Ramda` :

```
import * as _ from 'ramda';
```

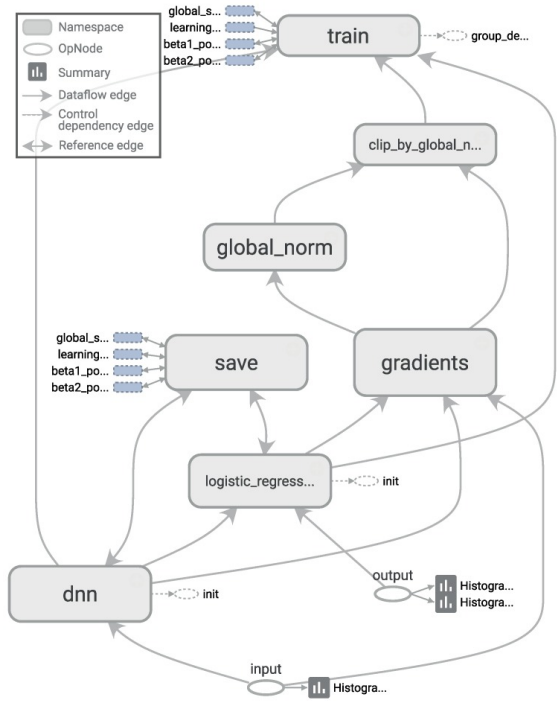
```
const addOrRemove = val ⇒                // Example adapted from R.W. Pearce
    _.ifElse(
        _.includes(val),                 // (arr) ⇒ is val in arr ?
        _.without(_.of(val)),            // (arr) ⇒ arr without [val]
        _.append(val)                    // (arr) ⇒ arr with val appended
);

addOrRemove(7)([1,2,3,4]); // → [ 1, 2, 3, 4, 7 ]
addOrRemove(3)([1,2,3,4]); // → [ 1, 2, 4 ]
```

- Avec `Ramda`, toutes les fonctions sont curryfiées par défaut.
- La curryfication simplifie la composition des fonctions.

Data Flow Programming

- Le style de programmation ainsi obtenu est orienté données (*data flow*) plutôt que contrôle (*control flow*).

- Il existe un paradigme de programmation orienté autour des flots de données (*dataflow programming*), structurés par un graphe.
- En particulier utilisé dans la bibliothèque `tensorflow` en apprentissage automatique.
- À contraster avec le style impératif classique.



Source : Y. Saez et. al.
DOI : 10.3390/s17010066

Map-Reduce

Les calculs sur les tableaux se chaînent facilement :

```
const data = [ { desc: 'Chocolate',  amount: 3, unitPrice: 10,  type: 'expense' },
               { desc: 'Tournament', amount: 1, unitPrice: 50,  type: 'asset' },
               { desc: 'Candy',      amount: 7, unitPrice: 3,   type: 'expense' },
               { desc: 'Lottery',    amount: 1, unitPrice: 100, type: 'asset' } ];
```

Version impérative

```
let total = 0;
for (let i = 0; i < data.length; i++) {
  if (data[i].type === 'expense'){
    let expense = data[i].amount *
                  data[i].unitPrice;
    total += expense;
  }
}
```

Version fonctionnelle / composition

```
const total = data
  .filter((line) => {
    return line.type === 'expense';
}).map((line) => {
    return line.amount *
           line.unitPrice;
}).reduce((sum, expense) => {
    return sum + expense; });
```

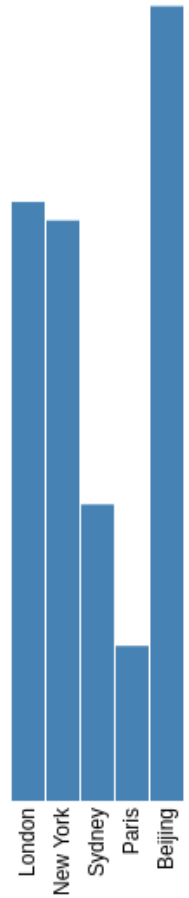
- Peut perdre en performance si chaque objet intermédiaire est construit.
- Pour mitiger cela, les calculs peuvent s'effectuer de manière **paresseuse**.

Fluent interfaces

Le chaînage peut devenir une propriété de l'interface de programmation, comme dans le cas des **fluent interfaces**.

L'exemple suivant est réalisé en `D3.js` :

```
const cities = [
        { name: 'London',   population: 8674000},
        { name: 'New York', population: 8406000},
        // ...
                ];
d3.selectAll('rect')
  .data(cities)
  .attr('height', 19)
  .attr('width', function(d) {
    var scaleFactor = 0.00004;
    return d.population * scaleFactor;
  })
  .attr('y', function(d, i) {
    return i * 20;
  });
```



Source : https://www.d3indepth.com/datajoins

Promises API

Les **promesses** (*promises*) permettent de chaîner des calculs asynchrones :

```
getJSON('https://web.site/story.json').     // Example from https://web.dev/promises
    then(function(story) {
      return getJSON(story.chapterUrls[0]);
    }).
    then(function(chapter1) {
      addHtmlToPage(chapter1);
    }).
    catch(function(error) {
      addTextToPage(`Failed to show chapter : ${error}`);
    }).
    then(function() {
      document.querySelector('.spinner').style.display = 'none';
    });
```

- Il faut voir ici les promesses comme une manière de **préparer un calcul**. Comme lorsqu'on compose des fonctions ou séquence des instructions.

L'inévitable slide sur les monades

La composition des fonctions peut être facilité par des choix de syntaxe.
L'exemple suivant en Haskell est un code fonctionnel pur :

```
battle = do                                   -- Adapted from www.haskellforall.com
    forM_ ["Take␣that!", "and␣that!", "and␣that!"] $ \taunt → do -- Charge!
        lift $ putStrLn taunt
        boss.health -= 10

    fireBreath (Point 0.5 1.5)                -- The dragon awakes !

    lift $ putStrLn "Retreat!"                -- Retreat is the better part of valor
    zoom (units.traversed.position) $ do
        x += 10                               -- Warning : this code is not 'vanilla' Haskell
        y += 10                               -- It makes usage of monads and lenses
```

Exemple tiré du blog Haskell For All

- Les calculs effectués ici sont des compositions de fonctions pures.
- Elles se transmettent un état initial et renvoient un état final.

Conclusion sur la 1ère classe

En suivant les idées de la 1ère classe, nous avons étudié des manières particulières d'utiliser les fonctions dans des structures :

- Les fonctions peuvent **représenter des valeurs complexes** en tant que tels (ensembles, prédicats, callbacks . . .) ;
- Si on les rassemble dans des structures, elles forment un début d'**architecture** (modules, objets . . .) ;

Un autre aspect remarquable tient dans la facilité à composer les fonctions.

- Les algorithmes peuvent être pensés comme des **compositions** de calculs, plutôt que des suites d'instructions (e.g data-flow).
- Les possibilités en terme de syntaxe et de forme d'API peuvent faciliter voire encourager un style fonctionnel.

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.