Programmation fonctionnelle

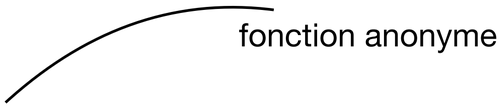
felipe@IFT1015 H2020

idée

- Beaucoup de traitements peuvent être vus comme l'application d'une fonction par un algorithme sur une structure de données.
    - en particulier, parcourir un tableau pour appliquer une fonction (ex: sqrt) à ses éléments est omniprésent dans nos codes
- Etre capable de **composer** des **fonctions** à la volée et les passer à un **algorithme** est une façon de voir la programmation qui peut s'avérer redoutablement efficace.
    - plus besoin notamment d'écrire de boucles
- Programmer fonctionnel requiert de connaître quelques algorithmes, quelques opérateurs sur les fonctions, et quelques fonctions de base.

Les fonctions = objet de premier ordre en javascript

1. On peut les stocker dans une structure




```
var f = fonction() {return 4;};
var t = [1,2,"trois", f, function(a,b) {return a+b;}];

assert(t[3]() === 4);
assert(t[4](2,3) === 5);
```

2. On peut les passer en paramètre d'une fonction

```
var f_arg2 = function(f,a,b) {return f(a,b);};
assert(f_arg2(t[4],5,6) === 11);
```

3. Peut être retourné par une fonction

```
var add = function(inc) {
    return function(a) {return a_+inc;};
};
var add3 = add(3);
assert( add3(2) === 5 );
assert( add(2)(4) === 6 );
```

fonction

fonction

Passer une fonction en paramètre

Très utile, par exemple pour les tris

```
arr.sort()
arr.sort(fonctionComparaison)
```

```
print([1,2,11,3,4,5,11,12].sort());
```
1,11,11,12,2,3,4,5 — ordre lexicographique

```
print([1,2,11,3,4,5,11,12].sort(function(a,b) {return a-b;}));
```
1,2,3,4,5,11,11,12 — ordre numérique croissant

```
print([1,2,11,3,4,5,11,12].sort(function(a,b) {return b-a;}));
```
12,11,5,4,3,2,1 — ordre numérique décroissant

La fonction passée doit retourner un nombre:
- 0 si les deux valeurs sont égales
- <0 si les deux valeurs sont dans le bon ordre
- >0 sinon

a-b vaut:
- 0 si a=b
- <0 si a < b
- >0 si a>b

note: sort (qui offre une implémentation du tri rapide) est dans la classe Array

Composer deux fonctions

$$f \circ g(x) = f(g(x))$$

```
var compose = function (f,g) {
    return function(x) {
       return f(g(x));
    };
}
```

fonction (anonyme) d'un argument

on pourrait implémenter la composition pour des fonctions d'arité quelqconque

```
assert(  compose(function(x) { return x * x; },
                 function(x) { return x + x; })(3) === 36 );

assert ( compose(function(x) { return x + x; },
                 function(x) { return x * x; })(3) === 18 );
```

bien sûr, on pourrait faire un fonction particulière pour calculer ceci :

```
var f_127 = function(x) {
  return (x+x) * (x+x);
}
```

mais il va nous falloir écrire des fonctions nommées pour beaucoup de choses.

note: compose est un opérateur

Algorithme: map

- Applique une fonction à chaque élément d'un tableau et retourne un nouveau tableau avec le résultat.
- map existe dans la classe Array
- Voici un code pour en comprendre l'idée:

```
Array.prototype.map = function (foncteur) {
    var tab = new Array(this.length);
    for (var i = 0; i< this.length; ++i)
       tab[i] = foncteur(this[i]);

    return tab;
};
```

```
var t = [1,2,3,4,5];

print(t.map(function(x) {return x * x;})); // 1,4,9,16,25
print(t.map(function(x) {return x +2;})); // 3,4,5,6,7

print([[3,1,5], [7,8,9,1], [1,3,2,1]].map(function(x) {return x.sort();})); // quiz
```

Algorithme: filter

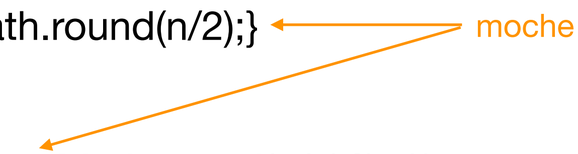
- Crée un nouveau tableau qui contient tous les éléments du tableau initial pour lesquels la fonction retourne true
- filter existe dans la classe Array
- Voici un code pour en comprendre l'idée:

```
Array.prototype.filter = function (foncteur) {
    var tab = [];
    for (var i = 0; i< this.length; ++i)
       if (foncteur(this[i]))
         tab.push(this[i]);

    return tab;
};
```

```
var t = [1,2,3,4,5,6,7];
var even = function(n) {return n / 2 === Math.round(n/2);}

print(t.filter(even)); // 2,4,6
print(t.filter(function(n) {return return n / 2 !== Math.round(n/2);}); // 1,3,5,7
print(t.filter(function(x) {return x < 5;}); // 1,2,3,4
```



D’autres opérateurs

```
/*
 *  negation d'une fonction (a un argument)
 */
var not = function(f) {

    return function(x) {
       return ! f(x);
    };
};

/*
 * fixer le second argument d'une fonction (a 2 arguments)
 */
var bind2nd = function(f,val) {

    return function (a) {
       return f(a,val);
    };
};

/*
 * f and g
 */
var and = function(f,g) {

    return function(x) {
       return f(x) && g(x);
    };
};
```

...

```
var odd  = not(even); // cool eh ?
t.filter(not(even));
```

```
var more = function(a,b) {return a > b;};
var less = function(a,b) {return a < b;};

t.filter(bind2nd(less,3)); // 1,2
```

fonction d’un argument qui retourne true s’il est inférieur à 3

```
t.filter(not(bind2nd(less,3))); // 4,5,6,7

t.filter(and(not(bind2nd(more,5)),
             not(even))); // 1,3,5
```

peut devenir rapidement difficile à lire (mais rien ne vous empêche de nommer les fonctions (ex: inf_ou_egal_a_5_et_impair).

and breathe normally

```
var t = ["jean", "michel", "fabienne", "paul"];

t.filter(function(e) {return e.length > 4;}); // ["michel", "fabienne"]

// de manière équivalente
t.filter(compose(bind2nd(more,4),function(e) {return e.length;})); // yeh

// ou encore
var length = function(x) {return x.length;};  // car utile
t.filter(compose(bind2nd(more,4), length)); // yeh
```

Algorithmes ++

- every retourne `true` si chaque élément du tableau satisfait la fonction de test passée en paramètre.
- every disponible dans la classe Array.
- voici un code possible

```
Array.prototype.every = function(predicat) {
    return this.filter(not(predicat)).length === 0;
};
```

```
assert([2,4,6,8,10].every(even));
```

- some retourne `true` si au moins un élément du le tableau satisfait la fonction de test passée en paramètre.
- some est dans Array
- voici un code possible

```
Array.prototype.some = function(predicat) {
    return this.filter(predicat).length > 0;
}
```

```
assert([1,4,6,8,10].some(even));
```

Algorithme: réduction

- reduce applique une fonction sur un accumulateur et sur chaque valeur du tableau (de gauche à droite) de façon à obtenir une unique valeur à la fin.
- reduce est dans Array
- en voici un code possible

```
Array.prototype.reduce = function(func,accumulator) {
    for (var i=0; i<this.length; ++i)
       accumulator = func(this[i],accumulator);
    return accumulator;
};

assert( [1,2,3,4,5].reduce(function(a,b) {return a+b;},0) === 15);


// ou encore
var add = function(a,b) { return a+b; };
var mul = function(a,b) { return a*b; };

assert( [1,2,3,4,5].reduce(add,0) === 15);
assert( [1,2,3,4,5].reduce(mul,1) === 120 );

// quiz
[1,2,3,4,5,6].reduce(function(a,b) {return b+1;}, 0);
```

l'implémentation par défaut attend une fonction de réduction de signature (accumulator, valeurCourante)

Algorithme: réduction

```
assert(["a","b","c","de","f"].reduce(add,"") === "abcdef");
```

note: avec l'implémentation vue:

```
Array.prototype.reduce = function(func,accumulateur) {
    for (var i=0; i<this.length; ++i)
       accumulateur = func(this[i],accumulateur);
    return accumulateur;
};
```

i	t[i]	accumulateur	opération
		""	
0	"a"	"a"	add("a","")
1	"b"	"ba"	add("b","a")
2	"c"	"cba"	add("c","ba")
3	"de"	"decba"	add("de","cba")
4	"f"	"fdecba"	add("f","decba")

note: reduceRight parcourt le tableau de la droite vers la gauche:

```
assert(["a","b","c","de","f"].reduceRight(add,"") === "fdecba");
```

Quiz

```
[[0, 1], [2, 3], [4, 5]].reduce(function(a, b) {
    return a.concat(b);
});
```

note: l'implémentation native prend la première valeur du tableau par défaut (premier pas de boucle à l'indice 1)

note: map-reduce est un patron d'architecture développé par Google qui permet le calcul parallèle massif

quicksort (le retour)

```
var quick = function (tab,comparateur) {
   if (tab.length <= 1) return tab;

   var pivot = tab.shift(); // not the best choice for a pivot
   var inf = quick(tab.filter(function(x) {return comparateur(x,pivot);}), comparateur);
   var sup = quick(tab.filter(not(function(x) {return comparateur(x,pivot);})), comparateur);
   inf.push(pivot);
   return inf.concat(sup);
};
```

note: pas efficace car on crée des tableaux à chaque pas de récursivité (et on prend comme pivot le premier élément), mais élégant !

```
print (quick([18,15,12,12,4,12,7,9,76,2], less).join(" "));   // 2 4 7 9 12 12 12 15 18 76
print (quick([18,15,12,12,4,12,7,9,76,2], more).join(" ")); // 76 18 15 12 12 12 9 7 4 2
```