Concepts fonctionnels

HOF, λ, closure, partiel et curryfication

Yannick Loiseau

API Hour – Clermont'ech

2014-02-26

http://yloiseau.net/articles/functional/

Conception

décomposition

en objet : classes → collaboration (top-down)

en fonctionnel : fonctions → composition (bottom-up)

fonction → valeur de 1er ordre

type (objet) *fonction*

- affectée à une « variable »
- passée en paramètre
- retournée

⇒ approches $\neq$

⇒ solutions $\neq$

Higher Order Function : fonction d'ordre supérieur

fonction

- paramètre : fonction
- retourne : fonction

⇒ abstraction d'algo.

doubler les valeurs d'une liste

```
def double_all(values) :
    result = []
    for val in values :
        result.append(2 * val)
    return result
```

```
assert double_all([1, 2, 3]) == [2, 4, 6]
```


ajouter un préfixe à une liste de chaînes

```
def hello_to_all(names) :
    result = []
    for name in names :
        result.append("Hello " + name + "!")
    return result
```

```
assert hello_to_all(["Julien", "William", "Yannick"]) == [
  "Hello Julien!",
  "Hello William!",
  "Hello Yannick!"
]
```



```
def double_all(values) :
    result = []
    for val in values :
        result.append(2 * val)
    return result
```

```
def hello_to_all(names) :
    result = []
    for name in names :
        result.append("Hello " + name + "!")
    return result
```

⇒ duplication algo. général

≠ calcul des nouvelles valeurs

⇒ fonction générique + fonction en paramètre

```
def apply_to_list(f, l) :
    result = []
    for elt in l :
        result.append(f(elt))
    return result
```

```
def double_me(v) :
    return 2 * v
```

```
assert apply_to_list(double_me, [1, 2, 3]) == [2, 4, 6]
```

```
def say_hello(name) :
    return "Hello " + name + "!"
```

```
assert apply_to_list(say_hello,
        ["J", "W", "Y"]) == ["Hello J!", "Hello W!", "Hello Y!"]
```


map

$$map : (a \to b) \times list\langle a \rangle \to list\langle b \rangle$$
$$map(f, [a_1, \ldots, a_n]) = [f(a_1), \ldots, f(a_n)]$$

produit des valeurs d'une liste

```
def prod_of(values) :
    result = 1
    for val in values :
        result = result * val
    return result
```

```
assert prod_of([1, 2, 3, 4]) == 24
```


« aplatir » une liste de listes

```
def flatten(lst) :
    result = []
    for sublst in lst :
        result = result + sublst
    return result
```

```
assert flatten(
    [
        ["a", "b"],
        ["c", "d", "e"],
        ["f"]
    ]
) == ["a", "b", "c", "d", "e", "f"]
```

```python
def prod_of(values) :
    result = 1
    for val in values :
        result = result * val
    return result
```

```python
def flatten(lst) :
    result = []
    for sublst in lst :
        result = result + sublst
    return result
```

⇒ duplication algo. général

≠ fonction d'agrégation, valeur initiale (elt. neutre)

⇒ fonction générique + paramètres


```python
def aggreg_by(f, z, l) :
    result = z
    for elt in l :
        result = f(result, elt)
    return result
```

```python
from operator import mul, add
```

```python
assert aggreg_by(mul, 1, [1, 2, 3, 4]) == 24
```

```python
assert aggreg_by(add, [],
    [
        ["a", "b"],
        ["c", "d", "e"],
        ["f"]
    ]
) == ["a", "b", "c", "d", "e", "f"]
```


fold (foldl) / reduce

map ? reduce ? *map* + *reduce* = MapReduce (1958)

$$foldl : (a \times b \to a) \times a \times list\langle b\rangle \to a$$
$$foldl(f, z, [a_1, \ldots, a_n]) = f(\ldots f(f(z, a_1), a_2), \ldots)$$

foldr

$$foldr : (a \times b \to b) \times b \times list\langle a\rangle \to b$$
$$foldr(f, z, [a_1, \ldots, a_n]) = f(a_1, f(a_2, f(\ldots, f(a_n, z))\ldots))$$

catamorphisme (th. des catégories)

$\circ : (b \to c) \times (a \to b) \to (a \to c)$
$(f \circ g)(x) = f(g(x))$

$foldr(cons, [], l) = copy(l)$

$foldl + cons \equiv reverse$

$foldr(cons \circ f, [], l) = map(f, l)$

$filter : (a \to boolean) \times list\langle a \rangle \to list\langle a \rangle$

...

D.I. / factory $\Rightarrow$ fonction (classe)

D.P. comportement $\to$ délégation/redéfinition

- stratégie
- décorateur
- *template method*
- commande

$\Rightarrow$ composition

automate (état, événements) $\Rightarrow$ Map$\langle$evt, fonction$\rangle$

visiteur $\Rightarrow$ fold

No More Patterns !

Peter Norvig $\to +\frac{2}{3}$ D.P. classiques (GoF)

D.P. commande : réduction par la composition

```
def command(*functions) :
    return reduce(compose, reversed(functions), ident)
```

```
def double(i) :
    return 2 * i

def add4(i) :
    return i + 4

def prefix(s) :
    return "val : " + s
```

```
c = command(double, add4, str, prefix)

assert c(19) == "val : 42"
```

```
d = command(int, add4, double)

assert d("17") == 42
```

$\equiv$ | Unix

✓ abstraction , réutilisation , simplification

✗ → fonctions *ad hoc* : utilisation unique

Lambda

λ-calcul

fonction anonyme

- à la demande
- locale
- récupérée (GC)


```
def double_me(v) :
    return 2 * v
```

```
double_me = lambda x : 2*x
```



```
assert map(lambda x : 2*x, [1, 2, 3]) == [2, 4, 6]
```

```
def hello_to_all(names) :
    return map(lambda s : "Hello " + s + "!", names)
```

```
assert hello_to_all(["J", "W", "Y"]) == ["Hello J!", "Hello W!", "Hello Y!"]
```

`Comparator<T>` → lambda

- tripler les valeurs de la liste ?
- dire « Goodbye » ?

autres lambda... même structure

⇒ généralisons plus !

Closure

« capture » valeurs → déf. de fonction

dual de l'objet

- objet : structure de donnée + comportements (fonctions)
- closure : fonction + données

⇒ famille de fonctions

Comment ?

fonction → lambda

⇒ *créer* des fonctions paramétrées

```
def say(pref, suff="") :
    return lambda name : pref + name + suff
```

```
say_hello = say("Hello ","!")
assert say_hello("Yannick") == "Hello Yannick!"
```

```
assert say("Farewell ")("Yannick") == "Farewell Yannick"
```

```
assert map(say("Hello ", "!"), ["J", "W", "Y"]) == [
                    "Hello J!",
                    "Hello W!",
                    "Hello Y!"
]
```

```
assert map(say("Goodbye ", "..."), ["J", "W", "Y"]) == [
                    "Goodbye J...",
                    "Goodbye W...",
                    "Goodbye Y..."
]
```


Runnable, Callable<V>, ...

classes anonymes → 1 méthode

⇒ closure

cas particulier (fréquent)

fonction de n paramètres → fonction de $n-1$ paramètres

composition

Application partielle

- fonction existante
- « fixer » paramètres
- → nouvelle fonction

⇒ closure → fonction générique

```
def partial(f, x) :
    return lambda *args : f(x, *args)
```

```
double = partial(mul, 2)
triple = partial(mul, 3)
plus5 = partial(add, 5)
```

```
assert double(21) == 42
assert triple(14) == 42
assert plus5(37) == 42
```


fixer plusieurs ?

```
def calcul(a, b, c, d) :
    return (a * b) + c - d
```

```
foo = partial(partial(partial(calcul, 7), 5), 12)

assert foo(5) == 42
```

hum...

```
assert reduce(partial, [7, 5, 12], calcul)(5) == 42
```

✓ composition, HOF

```
foo = compose(
        compose(
            partial(partial(foldl, mul), 1),
            partial(filter, even)
        ),
        partial(map, partial(add, 3))
)
assert foo([1,2,3,4,5]) == 192
```

✗ lourd...

Fonction Curryfiées

Haskell Curry

fonction n paramètres $\rightarrow$ série de fonction 1 paramètre

autom. partialisable

$add : int \times int \to int$

```
add = lambda a, b : a+b
```

$add : int \to (int \to int)$

```
add = lambda a : lambda b : a+b
```

```
plus10 = add(10)

assert plus10(32) == 42
```

```
assert add(21)(21) == 42
```


✗ pas naturel

Haskell, OCaml,. . . ⇒ auto-curryfiées (sans parenthèses)

```
foo = foldl (*) 1 . filter even . map (+3)
```

et les autres langages ?

fonction de « curryfication »

```
def curry(f) :
    return lambda a : lambda b : f(a, b)
```

```
from operator import mul

assert mul(21, 2) == 42
```

```
mul = curry(mul)

assert mul(21)(2) == 42
```


Et aussi

pour les curieux, ou une prochaine fois

- fonction pure, immutabilité
- idempotence
- transparence référentielle
- *memoize*, parallélisation, réparti
- évaluation stricte ou paresseuse
- *lazy list*, générateurs
- future, *continuation passing*
- homoiconicité, macro (lisp)
- DSL
- effets de bord, monades
- type option
- *pattern matching*
- structures récursives
- récursion terminale
- . . .