



Généralités



Anonymat



Arguments



Retours



Structures

# Approches Fonctionnelles de la Programmation

## ~ Fonctions d'Ordre Supérieur ~

Didier Verna

didier@didierverna.net



[didierverna.net](http://didierverna.net)



[@didierverna](https://twitter.com/didierverna)



[didier.verna](https://facebook.com/didier.verna)



[in/didierverna](https://in.linkedin.com/company/didierverna)

 **Plan**

Généralités

Fonctions Anonymes

Arguments Fonctionnels

Retours Fonctionnels

Structures de Données Fonctionnelles

 **Plan**

Généralités

Fonctions Anonymes

Arguments Fonctionnels

Retours Fonctionnels

Structures de Données Fonctionnelles

# Rappel : Ordre Supérieur

## ► Christopher Strachey

- nommage (variables)
- agrégation (structures)
- manipulation anonyme
- argument de fonction
- retour de fonction
- construction dynamique
- ...

# Définition, Nommage, Affectation

## ► Approche classique

### Lisp

```
(defun backwards (lst) (reverse lst))
```

### Haskell

```
backwards :: [a] -> [a]  
backwards xs = reverse xs
```

## ► Ordre supérieur

### Lisp (Lisp-2 !)

```
;; From function to function:  
(setf (symbol-function 'backwards) #'reverse)  
  
;; From variable to variable:  
(setq function2 function1)  
  
;; From function to variable:  
(setq function2 #'function1)  
  
;; From variable to function:  
(setf (symbol-function 'function2) function1)
```

### Haskell

```
backwards :: [a] -> [a]  
backwards = reverse
```

### Scheme (Lisp-1)

```
(define backwards reverse)  
(set! function2 function1)
```

## ► Agrégation (struct, data, etc.)

# Variables / Fonctions Locales

## ► Variables locales

### Lisp

```
(defun f (x)
  (let ((a (* x x))
        (b (+ (* x x) 1)))
    (+ (/ a b) (/ b a))))
```

```
(defun f (x)
  (let* ((a (* x x))
         (b (+ a 1)))
    (+ (/ a b) (/ b a))))
```

### Haskell

```
f :: Float -> Float
f x = let a = x * x
      b = a + 1
      in a / b + b / a
```

```
f :: Float -> Float
f x = a / b + b / a
  where a = x * x
        b = a + 1
```

⚠ **Lisp** : pas de références mutuelles dans `let` (utiliser `let*`)

# Variables / Fonctions Locales

## ► Fonctions locales

- **Haskell** : let et where
- **Lisp** : flet / labels (Lisp-2)

### Lisp

```
(defun ssq (x y)
  (flet ((square (x) (* x x)))
    (+ (square x) (square y))))
```

### Haskell

```
ssq :: Float -> Float -> Float
ssq x y = let square x = x * x
          in square x + square y
```

```
ssq :: Float -> Float -> Float
ssq x y = square x + square y
  where square x = x * x
```

# Application Partielle / Coupure (Haskell)

## Haskell

```
multiply :: Float -> Float -> Float
multiply a b = a * b

-- multiply a b
```

- ▶  $\rightarrow$  est associatif à droite  
 $\text{Float} \rightarrow (\text{Float} \rightarrow \text{Float})$
- ▶ L'application est associative à gauche:  
 $\text{multiply } 2 \ 5 \Leftrightarrow (\text{multiply } 2) \ 5$
- ▶ **Curryfication** : Les fonctions Haskell sont unaires
  - ▶ **Application partielle** :  
 $\text{multiply } 2 :: \text{Float} \rightarrow \text{Float}$
  - ▶ **Coupure d'opérateur** :  
 $(+2) \ (>3) \ (3:) \ (\text{"error: "++}) \ \text{etc.}$



# Application : nrsqrt II, le Retour

## Haskell

```
nrsqrt :: Float -> Float -> Float
-- nrsqrt x delta = nrfind 1.0 x delta
nrsqrt = nrfind 1.0

nrfind :: Float -> Float -> Float -> Float
nrfind yn x delta
  | nrfound yn x delta = yn
  | otherwise = nrfind (nrnext yn x) x delta

nrfound :: Float -> Float -> Float -> Bool
nrfound yn x delta = abs (x - yn * yn) <= delta

nrnext :: Float -> Float -> Float
nrnext yn x = (yn + x / yn) / 2
```

# Ordre Pseudo-Supérieur en Impératif

## C

```
typedef ... list;

list reverse (list l)
{
    /* ... */
}
/*
    new_lst = reverse (lst);
*/

typedef list (* list_to_list_f) (list);
list_to_list_f backwards = reverse;
/*
    new_lst = (* backwards) (lst);
    new_lst = backwards (lst);
*/
```



# Plan



Généralités

Fonctions Anonymes

Arguments Fonctionnels

Retours Fonctionnels

Structures de Données Fonctionnelles

# Fonctions Anonymes (« Lambda »)

## ► Principe

- Possibilité de *ne pas* nommer une fonction
- Utilisation littérale

## ► Dénotation

### Lisp

```
(lambda (x) (* 2 x))
```

### Haskell

```
\x -> 2 * x
```

## ► Application

```
((lambda (x) (* 2 x)) 4)
```

```
(\x -> 2 * x) 4
```

## ► Remarque : macro lambda (Lisp)

```
(lambda (x y ...) ...)  
=> (function (lambda (x y ...) ...))  
=> #'(lambda (x y ...) ...)
```

# Application aux Contextes Locaux

## ► Équivalence conceptuelle

### Lisp

```
(defun f (x)
  (let ((a (* x x))
        (b (+ (* x x) 1)))
    (+ (/ a b) (/ b a))))
```

```
(defun f (x)
  ((lambda (a b) (+ (/ a b) (/ b a)))
   (* x x) (+ (* x x) 1)))
```

### Haskell

```
f :: Float -> Float
f x = let a = x * x
      b = a + 1
      in a / b + b / a
```

```
f :: Float -> Float
f x = (\a b -> a / b + b / a)
      (x*x) (x*x+1)
```

⚠ **Remarque :** pas de références mutuelles possibles

# Application aux Contextes Locaux

## ► Imbrication

### Lisp

```
(defun f (x)
  (let* ((a (* x x))
        (b (+ a 1)))
    (+ (/ a b) (/ b a))))
```

```
(defun f (x)
  (let ((a (* x x))
        (let ((b (+ a 1)))
          (+ (/ a b) (/ b a)))))
```

```
(defun f (x)
  ((lambda (a)
    ((lambda (b)
      (+ (/ a b) (/ b a)))
     (+ a 1)))
   (* x x)))
```

### Haskell

```
f :: Float -> Float
f x = let a = x * x
      b = a + 1
      in a / b + b / a
```

```
f :: Float -> Float
f x = let a = x * x
      in let b = a + 1
         in a / b + b / a
```

```
f :: Float -> Float
f x = (\a ->
      (\b ->
        a / b + b / a)
      (a+1))
      (x*x)
```

# Pseudo-Anonymat en Impératif

## ► Blocs explicites

**C**

```
if (expression)
{
    /* Pseudo-anonymous procedure ... */
}
else
{
    /* Pseudo-anonymous procedure ... */
}
```

 **Plan**

Généralités

Fonctions Anonymes

Arguments Fonctionnels

Retours Fonctionnels

Structures de Données Fonctionnelles



# Application Fonctionnelle

- ▶ **Rappel :** Application /  $\beta$ -reduction ( $\lambda$ -calcul)
- ▶ apply et funcall

## Lisp

```
(+ 1 2 3 4 5)
(apply #' + '(1 2 3 4 5))
(apply #' + 1 2 3 '(4 5))
(funcall #' + 1 2 3 4 5)
```

# Abstraction (Manquante)

$$\sum_{n=a}^b f(n) = f(a) + \dots + f(b)$$

## Lisp

```
(defun sint (a b)
  (if (> a b)
      0
      (+ a (sint (1+ a) b))))
```

```
(defun ssq (a b)
  (if (> a b)
      0
      (+ (* a a) (ssq (1+ a) b))))
```

```
(defun spi (a b)
  (if (> a b)
      0
      (+ (/ 1.0 (* a (+ a 2)))
        (spi (+ a 4) b))))
```

## Haskell

```
sint :: Int -> Int -> Int
sint a b
  | a > b = 0
  | otherwise = a + sint (a+1) b
```

```
ssq :: Int -> Int -> Int
ssq a b
  | a > b = 0
  | otherwise = a*a + ssq (a+1) b
```

```
spi :: Int -> Int -> Float
spi a b
  | a > b = 0
  | otherwise = 1.0 / fromIntegral (a * (a+2))
               + spi (a+4) b
```

# Abstraction (Explicite)

## Lisp

```
(defun sigma (a b term next)
  (if (> a b)
      0
      (+ (funcall term a)
         (sigma (funcall next a) b term next))))
```

```
(defun sint (a b)
  (funcall #'sigma a b #'identity #'1+))

(defun ssq (a b)
  (funcall #'sigma a b (lambda (x) (* x x)) #'1+))

(defun spi-term (a)
  (/ 1.0 (* a (+ a 2))))
(defun spi-next (a)
  (+ a 4))
(defun spi (a b)
  (funcall #'sigma a b #'spi-term #'spi-next))
```

## Haskell

```
sigma :: Num a => Int -> Int
      -> (Int -> a) -> (Int -> Int) -> a
sigma a b term next
  | a > b = 0
  | otherwise = term a + sigma (next a) b term next
```

```
sint :: Int -> Int -> Int
sint a b = sigma a b id (+1)

ssq :: Int -> Int -> Int
ssq a b = sigma a b (\x -> x*x) (+1)

spitem :: Int -> Float
spitem a = 1.0 / fromIntegral (a * (a+2))

spinext :: Int -> Int
spinext a = a + 4

spi :: Int -> Int -> Float
spi a b = sigma a b spitem spinext
```

# Application : nrsqrt III, la Vengeance

$$\lim_{n \rightarrow +\infty} u_n(x) \quad \text{à } \delta \text{ près}$$

## Lisp

```
(defun limit (found next x delta)
  (limit1 1.0 found next x delta))

(defun limit1 (yn found next x delta)
  (if (funcall found yn x delta)
      yn
      (limit1 (funcall next yn x) found next x delta)))

(defun nrfound (yn x delta)
  (<= (abs (- x (* yn yn))) delta))

(defun nrnext (yn x)
  (/ (+ yn (/ x yn)) 2))

(defun nrsqrt (x delta)
  (limit #'nrfound #'nrnext x delta))
```

# Application : nrsqrt III, la Vengeance

$$\lim_{n \rightarrow +\infty} u_n(x) \quad \text{à } \delta \text{ près}$$

## Haskell

```
limit :: (Float -> Float -> Float -> Bool) -> (Float -> Float -> Float) -> Float -> Float -> Float
-- limit found next x delta = limit1 1.0 found next x delta
limit = limit1 1.0

limit1 :: Float -> (Float -> Float -> Float -> Bool) -> (Float -> Float -> Float) -> Float -> Float
--> Float
limit1 yn found next x delta
| found yn x delta = yn
| otherwise = limit1 (next yn x) found next x delta

nrfound :: Float -> Float -> Float -> Bool
nrfound yn x delta = abs (x - yn * yn) <= delta

nrnext :: Float -> Float -> Float
nrnext yn x = (yn + x / yn) / 2

nrsqrt :: Float -> Float -> Float
-- nrsqrt x delta = limit nrfound nrnext x delta
nrsqrt = limit nrfound nrnext
```

# Motif 1 : Mapping

► **Principe** : traiter tous les éléments d'une liste

► **Exemple**

## Lisp

```
(defun dbl (l)
  (if (null l)
      nil
      (cons (* 2 (first l)) (dbl (rest l)))))
```

► **Lisp** : (mapcar func list &rest lists)

► **Haskell** : `map :: (a -> b) -> [a] -> [b]`

► **Application**

```
(defun dbl (l)
  (mapcar (lambda (x) (* 2 x)) l))
```

## Haskell

```
dbl :: [Int] -> [Int]
dbl [] = []
dbl (x:xs) = 2*x : dbl xs
```

```
dbl :: [Int] -> [Int]
dbl = map (*2)
```

# Motif 1bis : Mapping Généralisé

► **Principe** : mapping sur plusieurs listes

► **Exemple**

## Lisp

```
(defun list+ (l1 l2)
  (if (or (null l1) (null l2))
      nil
      (cons (+ (first l1) (first l2))
            (list+ (rest l1) (rest l2)))))
```

► **Lisp** : mapcar variadique

► **Haskell** : `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`

► **Application**

```
(defun list+ (l1 l2) (mapcar #' + l1 l2))
```

## Haskell

```
(!+) :: [Int] -> [Int] -> [Int]
(!+) (x:xs) (y:ys) = (x + y) : xs !+ ys
(!+) _ _ = []
```

```
(!+) :: [Int] -> [Int] -> [Int]
(!+) = zipWith (+)
```

## Motif 2 : Filtrage

► **Principe** : conserver ou supprimer les éléments d'une liste

► **Exemple**

### Lisp

```
(defun pst (l)
  (cond ((null l) nil)
        ((> (first l) 0)
         (cons (first l) (pst (rest l))))
        (t (pst (rest l)))))
```

### Haskell

```
pst :: [Int] -> [Int]
pst [] = []
pst (x:xs)
  | x > 0 = x : pst xs
  | otherwise = pst xs
```

► **Lisp** : `(remove-if [-not] pred list &key ...)`

► **Haskell** : `filter :: (a -> Bool) -> [a] -> [a]`

► **Application**

```
(defun pst (l)
  (remove-if (lambda (x) (< x 0)) l))
```

```
pst :: [Int] -> [Int]
pst = filter (>0)
```



## Motif 3 : Folding / Réduction

- ▶ **Principe** : combiner tous les éléments d'une liste
- ▶ **Exemple**

### Lisp

```
(defun listsum (l)
  (cond ((null l) (error "empty list"))
        ((null (rest l)) (first l))
        (t (+ (first l) (listsum (rest l))))))
```

- ▶ **Lisp** : (reduce func seq &key ...)
- ▶ **Haskell** : foldr1 :: (a -> a -> a) -> [a] -> a
- ▶ **Application**

```
(defun listsum (l) (reduce #'+ l))
```

### Haskell

```
listsum :: [Int] -> Int
listsum [x] = x
listsum (x:xs) = x + listsum xs
```

```
listsum :: [Int] -> Int
listsum = foldr1 (+)
```

## Motif 3bis : Folding Généralisé

- ▶ **Principe** : folding avec valeur initiale (cas de la liste vide)
- ▶ **Lisp** : clé :initial-value de reduce
- ▶ **Haskell** : foldr

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f s []      = s
foldr f s (x:xs) = f x (foldr f s xs)
```

### ▶ Application

#### Lisp

```
; ; Nothing to do! (+) => 0
(defun listsum (l) (reduce #' + l))
```

#### Haskell

```
listsum :: [Int] -> Int
listsum = foldr (+) 0
-- sum !
```

# ssq II, le Retour

## ► Rappel : vision récursive

### Lisp

```
(defun ssq (n)
  (if (= n 1)
      1
      (+ (* n n) (ssq (1- n))))))
```

### Haskell

```
ssq :: Int -> Int
ssq 1 = 1
ssq n = n*n + ssq (n-1)
```

## ► Nouvelle vision : « foldmap » sur la liste des entiers

```
(defun ssq (n)
  (reduce #'+
    (mapcar (lambda (x) (* x x))
      (intlist n))))
```

```
ssq :: Int -> Int
ssq n = foldr1 (+) (map (\x -> x * x) [1..n])
-- ssq n = sum (map (\x -> x * x) [1..n])
```

# « Compréhenseurs » de Listes (Haskell)

$$\{f(x) \mid x \in E, C_1(x), C_2(x), \dots\}$$

- ▶ **Littéralement** : « l'ensemble des  $f(x)$  telle que  $x \in E$  et toutes les propriétés  $C_i(x)$  sont vérifiées »
- ▶ **Générateur** : `[ f | e <- l, c1, c2, ... ]`
- ▶ **Exemples**

```
[ 2*n    | n <- [2,3,4] ] -- [4,6,8]
[ n == 3 | n <- [2,3,4] ] -- [False, True, False]
[ 2*n    | n <- [2,3,4], n == 3 ] -- [6]
```

- ▶ **Avec pattern matching**

```
[ m*n | [m,n] <- [[2,3],[4,5],[6,7]], m > 2 ] -- [20, 42]
```

# Remarques sur les Compréhensions

## ► Utilisation comme filtre

```
import Char

digits :: String -> String
digits str = [ c | c <- str, isDigit c ]
```

## ► Utilisation en corps de fonction

```
allEven :: [Int] -> Bool
allEven xs = (xs == [e | e <- xs, isEven e])

allOdd :: [Int] -> Bool
allOdd xs = ([ ] == [e | e <- xs, isEven e])
```

## ⚠ Attention : variables locales aux compréhensions

```
bogusFind :: Int -> [[Int]] -> [[Int]]
bogusFind x ys = ([ x:zs | x:zs <- ys])
```

 **Plan**

Généralités

Fonctions Anonymes

Arguments Fonctionnels

Retours Fonctionnels

Structures de Données Fonctionnelles

# Généralités

- ▶ Principe évident : `#'+`, `func`, *etc.*
- ▶ Retours fonctionnels cachés
  - ▶ Application partielle : `(multiply 2)`
  - ▶ Coupure d'opérateur : `(+3)`
- ▶ Retour de fonctions existantes

## Lisp

```
(defun dispatch (obj)
  (typecase obj
    (class1 #'method1)
    (class2 #'method2)
    ...))
```

## Lisp

```
(defun +/- (plusp)
  (if plusp #'+' #-'))
;; (funcall (+/- t) 4 2)
```

## Haskell

```
plusOrMinus :: Bool -> (Int -> Int -> Int)
plusOrMinus True = (+)
plusOrMinus False = (-)
-- (plusOrMinus t) 4 2
```

## C

```
typedef int (* int_f) (int, int);
int_f plus, minus;

int_f plus_or_minus (int pls)
{
    return pls ? plus : minus;
}
/* (* plus_or_minus (1)) (4, 2); */
```

# Composition Mathématique ( $f \circ g$ )

## ► Exemple : complément logique

### Lisp

```
(defun isodd (n) (not (evenp n))) ;; oddp
```

```
(setf (symbol-function 'isodd) ;; oddp  
      (complement #'evenp))
```

### Haskell

```
isOdd :: Integer -> Bool -- odd  
isOdd n = not (even n)
```

```
isOdd :: Integer -> Bool -- odd  
isOdd = not . even
```

## ► Intérêt : orthogonalité

*Obsolescence des formes en `-not` (Lisp)*

```
(remove-if-not #'isodd lst)  
(remove-if (lambda (x) (not (evenp x))) lst)  
(remove-if (complement #'evenp) lst)
```

## ► Opérateur . (Haskell)

$(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$



# Composition Itérative ( $f^n$ )

## ► Récursion

### Haskell

```
iter :: Int -> (a -> a) -> (a -> a)
iter n f
  | n > 0 = f . iter (n - 1) f
  | otherwise = id
```

## ► Réduction

```
iter :: Int -> (a -> a) -> (a -> a)
iter n f = foldr (.) id (replicate n f)
```

# Retours Anonymes

## ► Complément logique

### Lisp

```
(defun complement (f)
  (lambda (&rest args) (not (apply f args))))
```

## ► Coupure d'Opérateurs / Application Partielle

```
(defun adder (n) (lambda (x) (+ x n)))
;; (funcall (adder 3) 1) => 4
```

### Haskell

```
adder :: Int -> (Int -> Int)
adder n = \x -> x + n
-- (adder 3) 1 => 4
```

## ► ssq III, la Vengeance

```
(defun foldmap (f m)
  (lambda (x) (reduce f (mapcar m x))))

(defun ssq (n)
  (funcall (foldmap #'* (lambda (x) (* x x)))
    (intlist n)))
```

```
foldmap :: (c -> c -> c) -> (c -> c)
         -> ([c] -> c)
foldmap f m = (foldr1 f) . (map m)

ssq :: Int -> Int
ssq n = foldmap (+) (\x -> x*x) [1..n]
```

 **Plan**

Généralités

Fonctions Anonymes

Arguments Fonctionnels

Retours Fonctionnels

Structures de Données Fonctionnelles

# Concept de « Donnée »

- ▶ **Vision impérative** (historique)
  - ▶ Représentation + Manipulation
  - ▶ Aggrégats + Fonctions
  - ▶ Classes + Méthodes
  - ▶ *etc.*
- ▶ **Abstraction de données**
  - ▶ Principe de génie logiciel
  - ▶ Distinguer l'interface de son implémentation
  - ▶ Toute implémentation peut convenir
- ▶ **Exemple**

**C**

```
complex_t make_complex (float real, float img);  
float real_part (complex_t complex);  
float img_part (complex_t complex);
```

**C++**

```
new Complex (float real, float img);  
float complex.real_part ();  
float complex.img_part ();
```



Implémentation (fonctionnelle (pure)) ?

# Concept de « Donnée »

## Lisp

```
(defun make-complex (real img)
  (lambda (selector)
    (case selector
      (:real real)
      (:img img))))

(defun real-part (complex)
  (funcall complex :real))

(defun img-part (complex)
  (funcall complex :img))
```

## Haskell

```
data Selector = Real | Img
type Complex = Selector -> Float

makeComplex :: Float -> Float -> Complex
makeComplex real img = select
  where select Real = real
        select Img = img

realPart :: Complex -> Float
realPart complex = complex Real

imgPart :: Complex -> Float
imgPart complex = complex Img
```

💡 Aucune structure de données / agrégat (objet = fonction)

► Stratégie d'envoi de message (« message passing »), cf. AOP !