Approches Fonctionnelles de la Programmation

Fonctions du 1er ordre

Didier Verna

didier@lrde.epita.fr
http://www.lrde.epita.fr/~didier

Table des matières

Définitions niveau fonction

Définitions simples

Lisp

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

Haskell

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

Scheme

```
(define (backwards lst)
  (reverse lst))
```

Lisp

```
(setf (symbol-function 'backwards)
      #'reverse)
```

Haskell

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

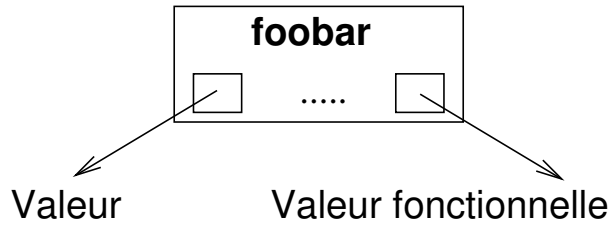
Scheme

```
(define backwards reverse)
```

Lisp : Symboles

Lisp-1 (Scheme) *vs.* Lisp-2 (Common Lisp)

- Les symboles contiennent des propriétés, dont une valeur et une valeur fonctionnelle.
- Un symbole peut dénoter à la fois une variable et une fonction (espaces de noms distincts).
- `defun` et `defparameter` ne sont pas indispensables (macros).




```
(defparameter *foo* 3)
(defun  *foo* (x) (* 2 x))
```

```
(setf (symbol-value '*foo*) 3)
(setf (symbol-function '*foo*)
      (lambda (x) (* 2 x)))
```

Lisp : Valeurs *vs.* valeurs fonctionnelles

API pour l'appel de fonction

- Fonction `function`, read-macro `#'`

```
(assert (eq (function +) #'+))
```

- Fonction `(funcall func &rest args)`

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

- Fonction `(apply func arg &rest args)`

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

Lisp : Fonctions variadiques

De l'influence de la syntaxe...

- Un appel de fonction Lisp est syntaxiquement clos :
  `(func arg1 arg2 ...)`
- **Fonctions variadiques :**

```
(defun mklist (head &rest tail)
  (cons head tail))
;; (mklist 'a 'b 'c)

(defun msg (str &optional (prefix "error: ") postfix)
  (concatenate 'string prefix str postfix))
;; (msg "hello" nil "!")

(defun msg* (str &key prefix (postfix "."))
  (concatenate 'string prefix str postfix))
;; (msg* "hello" :prefix "Me: ")
```

- Plus &-combinaisons ...

Haskell : Application partielle et coupure

De l'influence de la syntaxe...


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

-- multiply a b
```

- `->` est associatif à droite
  `Float -> (Float -> Float)`
- L'application est associative à gauche :
  `multiply 2 5` ⇔ `(multiply 2) 5`
- **Curryfication :** Les fonctions Haskell sont unaires
  - **Application partielle :**
    `multiply 2 :: Float -> Float`
  - **Coupure d'opérateur :**
    `(+2)` `(>3)` `(3:)` `("error: "++)` *etc.*

Fonctions anonymes : rappel

Des littéraux comme les autres...

- **Possibilité de *ne pas* nommer les fonctions :**

Lisp

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

Haskell

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

- **Utilisation directe (littérale) :**
  au même titre que les `int`, les chaînes de caractères *etc.*

Lisp

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

Haskell

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

Contextes locaux et fonctions anonymes

Équivalence conceptuelle

- `let` ouvre un bloc explicite
- Les fonctions ouvrent des blocs implicites
- Grâces aux fonctions anonymes, `let` est inutile

Lisp

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

Lisp

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

- **Remarques :**
  - Cela explique pourquoi les références croisées sont impossibles dans un `let`.
  - Modèle inapplicable à Haskell.

Application à `let*`
Imbrication

- `let*` se comporte comme des `let` imbriqués.

Lisp

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

Lisp

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

Lisp

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

Motivation (1)
Des formes communes d'abstraction

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

Lisp

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

```
(defun ssq (a b)
  (if (> a b)
      0
      (+ (sq 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 = sq 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
```

Motivation (2)

Des formes communes d'abstraction

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
           #'sq #'1+))

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

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 sq (+1)

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

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

spi :: Int -> Int -> Float
spi a b = sigma a b piterm pistep
```

Mapping
1er motif incontournable

- **Motif :**

Lisp

```
(defun double (lst)
  (if (null lst)
      nil
      (cons (* 2 (car lst))
            (double (cdr lst)))))
```

Haskell

```
double1, double2 :: [Int] -> [Int]

double1 [] = []
double1 (x:xs) = 2*x : double1 xs

double2 xs = [ 2*elt | elt <- xs ]
```

- **Lisp :** `(mapcar func list &rest lists)`
- **Haskell :** `map :: (a -> b) -> [a] -> [b]`
- **Exemples :**

Lisp

```
(defun double (lst)
  (mapcar (lambda (x) (* 2 x))
          lst))
```

Haskell

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

Mapping généralisé
Sur plusieurs listes

- **Motif :**

Lisp

```
(defun list+ (l1 l2)
  (if (or (null l1) (null l2))
      nil
      (cons (+ (car l1) (car l2))
            (list+ (cdr l1) (cdr l2)))))
```

Haskell

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

- **Lisp :** `mapcar` variadique.
- **Haskell :**

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

- **Exemples :**

Lisp

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

Haskell

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

Filtrage / Élimination

2e motif incontournable

- **Motif :**

Lisp

```
(defun pst (l)
  (cond ((null l)
         nil)
        ((> (car l) 0)
         (cons (car l)
               (pst (cdr l))))
        (t
         (pst (cdr l)))))
```

Haskell

```
pst1, pst2 :: [Int] -> [Int]
pst1 [] = []
pst1 (x:xs)
    | x > 0 = x : pst1 xs
    | otherwise = pst1 xs

pst2 xs = [ e | e <- xs, e > 0 ]
```

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

```
filter :: (a -> Bool) -> [a] -> [a]
```

- **Exemples :**

Lisp

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

Haskell

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

Folding / Réduction

3e motif incontournable

- **Motif :**

Lisp

```
(defun listsum (l)
  (cond ((null l)
         (error "empty list"))
        ((null (cdr l))
         (car l))
        (t (+ (car l)
              (listsum (cdr l))))))
```

Haskell

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

- **Lisp :** `(reduce func seq &key ...)`
- **Haskell :**

```
foldr1 :: (a -> a -> a) -> [a] -> a
```

- **Exemples :**

Lisp

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

Haskell

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

Folding généralisé

Valeur de retour pour la liste vide

- **Lisp :** clé `:initial-value` de `reduce`
  Remarque : `(+) => 0`, `(*) => 1`
- **Haskell :** `foldr`

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

- **Folding généralisé et récursion primitive :**
  Beaucoup de fonctions primitive-récursives peuvent s'exprimer comme un fold.

Lisp

```
(defun snoc (elt lst)
  (append lst (list elt)))

(defun rev (lst)
  (reduce #'snoc lst
          :from-end t
          :initial-value nil))
```

Haskell

```
snoc :: a -> [a] -> [a]
snoc x xs = xs ++ [x]

rev :: [a] -> [a]
rev = foldr snoc []
```

Recherche / indexation

- **Recherche :**
  - Lisp : `(member-if[-not] pred lst &key ...)` (réservé aux listes),
    `(find-if[-not] pred seq &key ...)`
  - Haskell :
    ```
    find :: (a -> Bool) -> [a] -> Maybe a
    ```
- **Indexation :**
  - Lisp :
    `(position-if[-not] pred seq &key ...)`
  - Haskell :
    ```
    findIndex :: (a -> Bool) -> [a] -> Maybe Int
    ```

Extraction

- **Motif :**

Haskell

```
getWord :: String -> String
getWord [] = []
getWord (x:xs)
    | x == ' ' = []
    | otherwise = x : getWord xs
```

- **Haskell :**

```
takeWhile :: (a -> Bool) -> [a] -> [a]
dropWhile :: (a -> Bool) -> [a] -> [a]
```

- **Exemples :**

Haskell

```
getWord :: String -> String
getWord str = dropWhile (\x -> x == ' ') str
```

Divers
Comme d'été

- **Prédicats :**
  - Lisp : Fonctions
    `(every|some pred seq &rest seqs)`
  - Haskell : Fonctions
    `all|any :: (a -> Bool) -> [a] -> Bool`
- **Tri :**
  - Lisp : Fonction `(sort seq pred &key ...)`
    Attention : `sort` est destructif !

```
(setf foo (sort #'> foo))
```

  - Haskell : Fonctions

```
sort :: Ord a => [a] -> [a]
sortBy :: (a -> a -> Ordering) -> [a] -> [a]
```

Motivation
Une question d'orthogonalité

- Fonctions complémentaires de Lisp :

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

  - Fonction `complement` : -50% de prédicats (obsolescence des formes en `-not`)

- Macro `setf` : accesseurs en tant que lvalues...

```
(defvar foo '(1 2 3 4 5))
(nth 2 foo) ;; => 3
(setf (nth 2 foo) 0) ;; setnth doesn't exist
foo ;; => (1 2 0 4 5)
```

Lisp : Retours fonctionnels et scoping

Problème de la « capture » de variables

Fonctions constantes :

```
(defun +/- (which_one)
  (if (= which_one 0)
      #'+
      #'-))

;; (funcall (+/- 0) 4 2)
```

Fonctions à la demande :

```
(defun make-adder (n)
  #'(lambda (x)
      (+ x n)))

;; (funcall (make-adder 3) 1)
```

- **Scoping dynamique :** retour fonctionnel limité aux fonctions constantes
- **Scoping lexical :** retour de fonctions créés à la demande en toute sécurité

```
(defun complement (fn)
  #'(lambda (&rest args)
      (not (apply fn args))))
```

- **Remarque :** retour fonctionnel et fermetures lexicales = coût principal d'implémentation du 1er ordre.

Haskell : Rappels
Retours fonctionnels cachés

- **Coupure d'opérateur / application partielle :**

```
(+3) :: Int -> Int
```

- **Définition niveau fonction :**

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

-- (adder 3) 1
```

```
adder :: Int -> (Int -> Int)
adder n = add
          where add x = x + n

-- (adder 3) 1
```

Composition de fonctions

Lisp

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

```
(defun compose (&rest fns)
  ;; Not trivial. Only the first
  ;; function can be variadic.
  )

(defun isodd (n)
  (compose #'not #'evenp))
```

Haskell

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

```
isodd :: Integer -> Bool
isodd = not . even
```

- `complement` : cas particulier de `compose`
- **Type Haskell :**

  `(.) :: (b -> c) -> (a -> b) -> (a -> c)`
- **Associativité de . :**

  `f . (g . h) = (f . g) . h = f . g .h`
- **Précédence Haskell :** `f . g x = f . (g x)`

Composition itérative

$f^n(x)$

- **Vision récursive :**

Lisp

```
(defun iter (n f)
  (if (> n 0)
      (compose f (iter (1- n) f))
      #'identity))
```

Haskell

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

- **Vision `fold`'esque :**

```
(defun iter (n f)
  (reduce #'compose (make-list n :initial-element f)
          :initial-value #'identity
          :from-end t))
```

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

(Dé) Curryfication

Arité des fonctions

- **Formes (dé) curryfiée :**

Haskell

```haskell
multiply :: Int -> Int -> Int
multiply a b = a * b
```

Haskell

```haskell
multiply :: (Int, Int) -> Int
multiply (a, b) = a * b
```

- **Haskell : (Dé) Curryfication**

```haskell
curry :: ((a, b) -> c)
      -> (a -> b -> c)
curry g x y = g (x, y)
```

```haskell
uncurry :: (a -> b -> c)
        -> ((a, b) -> c)
uncurry g (x, y) = g x y
```

- **Lisp : Currification partielle**

Lisp

```lisp
(defun curry (fn &rest args)
  #'(lambda (&rest args2)
      (apply fn
             (append args args2))))
```

Lisp

```lisp
(defun rcurry (fn &rest args)
  #'(lambda (&rest args2)
      (apply fn
             (append args2 args))))
```

Récurseurs de listes
Version initiale

Motif :

```
(defun len (lst)
  (if (null lst)
      0
      (1+ (len (cdr lst)))))
```

```
len :: [a] -> Int
len [] = 0
len (x:xs) = 1 + len xs
```

Récurseur :

```
(defun lrec (fn val)
  (labels ((self (lst)
             (if (null lst)
                 val
                 (funcall fn (self (cdr lst))))))
    #'self))
;; (funcall (lrec #'1+ 0) '(1 2 3 4 5)) => 5
```

```
lrec :: (a -> a) -> a -> ([b] -> a)
lrec fn val = let self [] = val
                  self (x:xs) = fn (self xs)
                  in self
-- (lrec (1+) 0) [1,2,3,4,5] => 5
```

Récurseurs de listes
Version générale

- **Motif :**

```
(defun every1 (p lst)
  (if (null lst)
      t
      (and (funcall p (car lst))
           (every1 p
                   (cdr lst)))))
```

```
all1 :: (a -> Bool) -> [a] -> Bool
all1 _ [] = True
all1 p (x:xs) = p x
                && all1 p xs
```

- **Récurseur :**

```
(defun lrec1 (fn val)
  (labels ((self (lst)
             (if (null lst)
                 val
                 (funcall fn (car lst)
                          #'(lambda ()
                              (self (cdr lst)))))))
    #'self))
```

```
lrec1 :: (a -> b -> b) -> b -> ([a] -> b)
lrec1 fn val = let self [] = val
                   self (x:xs) = fn x (self xs)
                   in self
```

Application
Récurseurs communs


```
;; every for some PREDicate
(lrec1 #'(lambda (elt cnt) (and (PRED elt) (funcall cnt))) t)

;; some for some PREDicate
(lrec1 #'(lambda (elt cnt) (or  (PRED elt) (funcall cnt))) nil)

;; find-if for some PREDicate
(lrec1 #'(lambda (elt cnt) (if (PRED elt) elt (funcall cnt))) nil)

;; remove-if-not for some PREDicate
(lrec1 #'(lambda (elt cnt) (if (PRED elt)
                               (cons elt (funcall cnt))
                               (funcall cnt))) nil)
```

```
-- all for some PREDicate
lrec1 (\elt cnt -> PRED elt && cnt) True

-- any for some PREDicate
lrec1 (\elt cnt -> PRED elt || cnt) False

-- find for some PREDicate
lrec1 (\elt cnt -> if PRED elt then Just elt else cnt) Nothing

-- filter for some PREDicate
lrec1 (\elt cnt -> if PRED elt then (elt : cnt) else cnt) []
```

Lisp : Récurseurs d'arbres

Représentation par liste hétérogène

- **Motifs :**

```
(defun leaves (tree)
  (if (atom tree)
      1
      (+ (leaves (car tree))
         (or (if (cdr tree) (leaves (cdr tree)))
             1))))
```

```
(defun flatten (tree)
  (if (atom tree)
      (list tree)
      (append (flatten (car tree))
              (if (cdr tree) (flatten (cdr tree))))))
```

Application

`leaves` et `flatten`

Récurseur :

```
(defun trec (fn val)
  (labels ((self (tree)
             (if (atom tree)
                 (funcall val tree)
                 (funcall fn
                          #'(lambda ()
                              (self (car tree)))
                          #'(lambda ()
                              (if (cdr tree)
                                  (self (cdr tree))))))))
    #'self))
```

Exemples :

```
;; leaves
(trec #'(lambda (lcnt rcnt)
          (+ (funcall lcnt) (or (funcall rcnt) 1)))
      #'(lambda (elt)
          1))

;; flatten
(trec #'(lambda (lcnt rcnt)
          (append (funcall lcnt) (funcall rcnt)))
      #'list)
```

Qu'appelle t'on « donnée » ?

- **En général :**
  - **Représentation :** structures de données
  - **Manipulation :** fonctions sur les données
- **Abstraction de données / interface :**
  - Constructeurs, accesseurs et manipulateurs
  - Conditions d' inter-fonctionnement
  - Toute implémentation doit satisfaire ces conditions
- **Conclusion :**
  - Distinguer l'interface de son implémentation
  - Toute implémentation peut convenir, y compris une implémentation fonctionnelle (fermetures lexicales) au lieu de structures de données

Lisp : Listes fonctionnelles
Application


```
(defun fcons (car cdr)
  (labels ((dispatch (arg)
             (cond ((= 0 arg)
                    car)
                   (t
                    cdr))))
    #'dispatch))

(defun fcar (obj)
  (funcall obj 0))

(defun fcdr (obj)
  (funcall obj 1))
```

- Les fonctions du 1er ordre (fermetures lexicales) permettent la représentation de données complexes.
- L'implémentation fonctionnelle utilise une stratégie d'« envoi de message » (message passing).