

Chapitre 3 : Fonctions (suite)

1. Récursivité terminale
2. Fonctionnelles de listes : map, filter & C°
3. Définitions locales : let & where
4. Polymorphisme. Type « le plus général » d'une fonction

1. Récursivité terminale — Technique de l'accumulateur

```
somme2 :: [Int] -> Int
```

```
somme2 l = somme2' 0 l
```

```
-- somme2' acc l = somme des éléments + acc(cumulateur)
```

```
somme2' :: Int -> [Int] -> Int
```

```
somme2' acc [] = acc
```

```
somme2' acc (x:xs) = somme2' (x+acc) xs
```

- Suite des appels récursifs

```
somme2 1 = somme2' 0 1
```

```
somme2' acc [] = acc
```

```
somme2' acc (x:xs) = somme2' (x+acc) xs
```

```
somme2 [1,2,3]
```

⇓

```
somme2' 0 [1,2,3]
```

```
-- décomposition x:xs = [1,2,3]
```

⇓

```
Somme2' 1+0 [2,3]
```

```
-- décomposition x:xs = [2,3]
```

⇓

```
Somme2' 2+1 [3]
```

```
-- décomposition x:xs = [3]
```

⇓

```
Somme2' 3+3 []
```

```
-- décomposition [] = []
```

⇓

6

- Récursivité non terminale

somme1 **[]** = 0

somme1 **x:xs** = **x** + (somme1 **xs**)

L'appel récursif est suivi d'une opération (x+...)

Récursivité non terminale

Empilement de contextes à l'évaluation

- Récursivité terminale

somme2 l = somme2' 0 l

somme2' acc **[]** = acc

somme2' acc (x:xs) = somme2' (x+acc) xs

L'appel récursif n'est suivi d'aucune opération

Récursivité terminale

Pas d'empilement de contextes à l'évaluation

==> Meilleure efficacité : gain de mémoire et de temps

==> Les compilateurs produisent du code optimisé

- Autres exemples, variantes

```
-- take
```

```
-- Forme non terminale
```

```
mytake3 0 _ = []
```

```
mytake3 _ [] = []
```

```
mytake3 k (x : xs) = x : (mytake3 (k-1) xs)
```

```
-- Récursivité terminale
```

```
mytake4 k l = mytake4' k [] l
```

```
mytake4' 0 acc _ = reverse acc
```

```
mytake4' k acc (x:xs) = mytake4' (k-1) (x:acc) xs
```

2. Fonctionnelles de listes

Un principe majeur de la **programmation fonctionnelle**:

- Les fonctions sont des objets « à part entière », qui peuvent être passés en argument d'une autre fonction et « synthétisées »
- On dit encore : « des citoyens de première classe »

Une **fonctionnelle** est une fonction prenant en argument / retournant des fonctions

Une première application : **fonctionnelles de listes**

1. Premières fonctionnelles

- `map` : (map f l) applique f à tous les éléments de la liste l

```
> :t map
```

```
map :: (a -> b) -> [a] -> [b]
```

```
> map carre [1..9]
```

```
[1,4,9,16,25,36,49,64,81]
```

```
> map head [ "Appellation", "Origine", "Contrôlée"  
"AOC"
```

```
> let succ x = x+1
```

```
> map succ [1..10]
```

```
[2,3,4,5,6,7,8,9,10,11]
```

```
> map odd [1..5]
```

```
[True,False,True,False,True]
```

- `filter` : (filter p l) sélectionne dans l les éléments qui satisfont p

```
> :t filter
```

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

```
> filter (>3) [1..10]
```

```
[4,5,6,7,8,9,10]
```

```
lesup x l = filter (>x) l
```

```
> lesup 3 [1..9]
```

```
[4,5,6,7,8,9]
```

- Autres : `foldr`...

2. Fonction anonyme (*lambda abstraction*)

- Maths :

$\lambda x.f(x)$ désigne la fonction $x \mapsto f(x)$

Exemples :

$\lambda x.x+1$ = fonction « successeur »

$\lambda x.x \geq 0$ = prédicat « positif ou nul »

NB. Distinguer de :

$f(x)$ = valeur de f en x (variable, inconnu)

$x+1$ = successeur de x

...

- Notation Haskell :

$\lambda x \rightarrow (f\ x)$

$\lambda x \rightarrow x+1$

$\lambda x \rightarrow x > 0$

- Exemples

- $\lambda x \rightarrow x+1$

fonction successeur : à x associe $x+1$

- $\lambda x \rightarrow x \Rightarrow 0$

prédicat « être positif »

- $\lambda x \rightarrow x * 2$

fonction « double »

- $\lambda x y \rightarrow [x,y]$

fabrique une liste à 2 éléments

- $\lambda x y \rightarrow (x+y)/2$

moyenne

> $(\lambda x y \rightarrow [x,y]) 2 3$

[2,3]

> $(\lambda x y \rightarrow (x+y)/2) 2 3$

2.5

> $(\lambda x y \rightarrow x > y) 2 3$

False

- Forme (presque) générale (1)

`(\ x1 x2... xn -> Exp)`

= la fonction qui à `x1 x2... xn` associe `Exp`

Evaluation :

`(\ x1 x2... xn -> Exp) arg1 arg2 ... argn`

Evalue `Exp` dans un environnement ou `xi` vaut `argi` pour tout `i`

- Application. Exemples;

Eviter de définir une fonction « de peu d'intérêt général »

```
> map (\x->x+1) [1..10]  
[2,3,4,5,6,7,8,9,10,11]
```

```
> filter (\x->(5<x && x<10)) [1..20]  
[6,7,8,9]
```

- Un « plus » : les lambdas acceptent le filtrage

```
> let ages =  
[ ("Jean",23), ("Marie",25), ("Paul",30), ("Anne",18) ]
```

```
-- les moins de 25 ans
```

```
> filter (\(nom,age) -> (age<=25)) ages
```

```
-- Les couples ordonnés
```

```
lesordonnes :: [(Int,Int)]->[(Int,Int)]
```

```
lesordonnes l = filter (\(x,y)->(x<y)) l
```

- Forme générale (2)

`(\ pat1 pat2... patn -> Exp)`

Evaluation :

`(\ pat1 pat2... patn -> Exp) x1 x2 ... xn`

Filtre chaque `xi` par le patron `pati` correspondant

Et évalue Exp avec les valeurs résultantes des variables

3. Application partielle d'une fonction

Question : quelle est la signification précise de

$a \rightarrow b \rightarrow c$?

Nous avons dit :

« Fonctions à 2 arguments de types a et b , à valeur dans c »

```
sommeCarres :: Float -> Float -> Float
sommeCarres x y = (carre x) + (carre y)
xor :: Bool -> Bool -> Bool
xor x y = (x || y) && not (x && y)
> xor True False
True
elem :: a -> [a] -> Bool
> elem 3 [1..9]
True
```

En fait : $f :: a \rightarrow b \rightarrow c$ est une fonction qui
pour tout $x :: a$
retourne une fonction $(f\ x) :: b \rightarrow c$

= « Application partielle » au premier argument

= Parenthésage implicite (associativité à droite)

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

```
modulo :: Int -> Int -> Int
```

```
modulo x y = mod y x
```

```
> modulo 3 7
```

```
1
```

```
> :t (modulo 3)
```

```
modulo 3 :: Int -> Int
```

```
> map (modulo 3) [1..12]
```

```
[1,2,0,1,2,0,1,2,0,1,2,0]
```

4. Application : Sections (opérateurs infixes)

- Notations infixe, préfixe, préfixe parenthésée, postfixe

Fonctions (mathématiques) binaires. Plusieurs notations possibles.

$2 + 3$, $3 * 5$, $2 + 3 * 5$ infixe : l'opérateur est entre les arguments
 $(2 + 3) * 5$

$+ 2 3$, $* 3 5$, $+ 2 * 3 5$ préfixe (polonaise)
 $(+ 2 3)$, $(* (+ 2 3) 5)$

$2 3 +$, $3 5 *$, $2 3 + 5 *$ postfixe

$f(2,3)$, $g(3,5)$, $h(2,3,5)...$ préfixe parenthésée (nb d'arg quelconque)

- Question : en haskell, quelle notation pour +, *, ^...

Réponse : infixée

```
> 2+3
```

```
5
```

```
> + 2 3
```

```
<interactive>:1:0: parse error on input `+'
```

```
> 5 < 4+3
```

```
True
```

- Comment récupérer la notation préfixée « standard » ?

```
> (+) 2 3
```

```
5
```

```
:t (+)
```

```
(+) :: forall a. (Num a) => a -> a -> a
```

```
> (<) 3 5
```

```
True
```

- Et le passage partiel d'arguments ?

```
> (+ 3) 2
```

```
5
```

```
> (< 3) 5
```

```
False
```

```
> map (*2) [1..5]
```

```
[2,4,6,8,10]
```

```
> map (\x -> x*2) [1..5]
```

```
[2,4,6,8,10]
```

```
> filter (>5) [1,6,2,3,8,9,0]
```

```
[6,8,9]
```

```
filter (\x -> x>5) [1,6,2,3,8,9,0]
```

```
[6,8,9]
```

(+) (>) (+2) (0<) ... sont appelés des *sections*

3. Définitions locale : Let et Where

- Définitions locales à une expression

On peut introduire une 'valeur', affectée à une variable, pour la réutiliser ensuite...

```
cherche5 x l =  
  if l == [] then False  
  else let y = (head l)  
        ys = (tail l)  
        in  
        (x == y) || (cherche5 x ys)
```

...ou définir cette variable *a posteriori*

```
Cherche6 x l =  
  if l == [] then False  
    else (x == y) || (cherche6 x ys)  
    where y = (head l)  
          ys = (tail l)
```

- Combinaison avec d'autres structures de contrôle

-- Combinaison let + gardes

```
cherche51 x l
  | l == []      = False
  | otherwise    = let  y = (head l)
                     ys = (tail l)
                     in (x == y || cherche51 x ys)
```

-- where + gardes

```
cherche52 x l
  | l == []      = False
  | otherwise    = (x == y || cherche52 x ys)
                  where y = (head l)
                        ys = (tail l)
```

- NB. Attention à l'indentation !

```

-- Combinaison let + filtrage
cherche53 x l
    | l == []      = False
    | otherwise    = let (y:ys) = l
                      in (x == y || cherche53 x ys)

-- where
cherche54 x l
    | l == []      = False
    | otherwise    = (x == y || cherche54 x ys)
                      where y:ys = l

```

- NB. Attention à l'indentation !

- Forme générale du let

```
let  pat1 = Exp1
    pat2 = Exp2
    ...
    patn = Expn
in Expression
```

- Forme générale du where

Expression

```
where pat1 = Exp1
      pat2 = Exp2
      ...
      patn = Expn
```

• Applications

- Utiliser deux fois une même expression sans la réévaluer
- Rend les définitions + lisibles en donnant un nom à des « valeurs clés »
- Fonctions « locales »

```
-- Normalisation d'un vecteur
```

```
type Vect = (Float,Float)
```

```
norme :: Vect -> Float
```

```
norme (x,y) = sqrt (x^2 + y^2)
```

```
normalise, normalise' :: Vect -> Vect
```

```
normalise (x,y)
```

```
    | norme (x,y) == 0           = (0,0)
```

```
    | otherwise                  = (x/norme (x,y), y/norme (x,y))
```

```
normalise' (x,y)
```

```
    | n == 0   = (0,0)
```

```
    | otherwise      = (x/n, y/n)
```

```
    where n = norme (x,y)
```

-- zip et unzip

```
> zip "abc" [1..3]
[('a',1),('b',2),('c',3)]
> unzip(zip "abc" [1..3])
("abc",[1,2,3])
```

```
myunzip :: [(a,b)] -> ([a],[b])
myunzip [] = ([],[ ])
myunzip ((x,y):reste) =
    let (xs,ys) = myunzip reste
    in (x:xs, y:ys)
```

```
myunzip2 :: [(a,b)] -> ([a],[b])
myunzip2 [] = ([],[ ])
myunzip2 ((x,y):reste) = (x:xs, y:ys)
    where (xs,ys) = myunzip2 reste
```

Définition de fonctions locales

- **Let** et **where** permettent aussi de définir des **fonctions locales**

Exemple : palindromes

abcacba abbccbba

in girum imus nocte et consumimur igni

elu par cette crapule

```
debut l = take ((length l)-1) l
```

```
palin3 l = let n = length l  
           in if n <= 1 then True  
              else head l == last l &&  
                  palin3 (debut (tail l))
```

```
palin1 l =  
    let debut l = take (length l - 1) l  
    in if length l <= 1 then True  
       else head l == last l &&  
           palin1 (debut (tail l))  
  
palin2 [] = True  
palin2 [_] = True  
palin2 (x:xs) =  
    (let debut l = take ((length l)-1) l  
     in (x == last xs) && palin2 (debut xs))
```

```
normalise'' (x,y)
| n == 0    = (0,0)
| otherwise  = (x/n, y/n)
where       norme (x,y) = sqrt (x^2 + y^2)
           n = norme (x,y)
```

- Autres exemples, variantes

-- somme « terminale » avec let/where pour une définition locale de fonction auxiliaire

```
somme3 l = somme3' 0 l
      where somme3' acc [] = acc
              somme3' acc (x:xs) = somme3' (x+acc) xs
```

```
somme4 l = let somme4' acc [] = acc
              somme4' acc (x:xs) = somme4' (x+acc) xs
            in somme4' 0 l
```

Portée de *let* et *where*

- Les valeurs et fonctions introduites ne sont définies que dans la portée de *let ... in* et *where*

```
> let x = 1 in x+1
```

```
2
```

```
> x
```

```
<interactive>:1:0: Not in scope: `x'
```

```
> let f x = x+1 in f 1
```

```
2
```

```
> f 1
```

```
<interactive>:1:0: Not in scope: `f'
```

Portée de *let* et *where*

- Mais *let seul* permet d'introduire valeurs et fonctions dans l'espace de travail

```
> let x =2
```

```
> x
```

```
2
```

```
> let succ x = x+1
```

```
> succ 5
```

```
6
```

4. Polymorphisme Type « le plus général » d'une fonction - Contrôle de type

- Pour toute fonction Haskell définit *le type de plus général possible*, à partir de sa « forme » et du type des fonctions appelées
- *Polymorphisme* : une fonction peut avoir plusieurs types — Ou : elle possède un *type polymorphe*
- *Haskell vérifie que la concordance entre le type d'une fonction et le type de ses arguments*

- Contrôle de type : A l'exécution

```
-- carre et quad
carre :: Int -> Int
carre x  = x^2
quad x = (carre x) ^2
> :t quad
???
```

```
> quad '2'
```

```
<interactive>:1:5:
```

```
    Couldn't match expected type `Int' against inferred type
    `Char'
```

```
    In the first argument of `quad', namely '2'
```

```
    In the expression: quad '2'
```

```
    In the definition of `it': it = quad '2'
```

- Permet un contrôle de type: Au cours de la définition

```
-- unzip (avec erreur de type)
myunzip :: (a,b) -> ([a],[b])
-- myunzip :: [(a,b)] -> ([a],[b])
myunzip [] = ([],[ ])
myunzip ((x,y):reste) =
    let (xs,ys) = myunzip reste
    in (x:xs, y:ys)
```

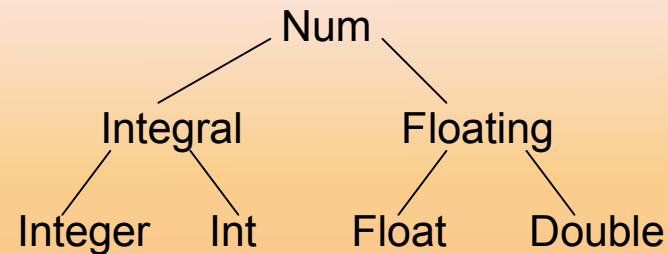
```
> :r
[1 of 1] Compiling Main                ( cours3.hs, interpreted )
cours3.hs:37:8:
    Couldn't match expected type `(a, b)' against inferred
    type `[a1]'
```

In the pattern: []

In the definition of `myunzip': myunzip [] = ([], [])

Failed, modules loaded: none.

- Une variable de type peut être astreinte à une « classe »



```

> :t (+)
(+) :: forall a. (Num a) => a -> a -> a
> :t \x y -> (div x y) == 0
\x y -> (div x y) == 0 :: forall a. (Integral a) => a -> a ->
  Bool

```

```

mystere 1 y = y
mystere x y = if (mod x 2 == 0)
  then mystere (div x 2) (y * 2)
  else y + mystere (div x 2) (y * 2)

```

- Un type peut être astreint à posséder certaines fonctions

```
-- insertion dans une liste triée
insere :: (Ord a) => a -> [a] -> [a]
insere x [] = [x]
insere x (y:ys) = if x<=y then x:y:ys else y:(insere x
    ys)
```

```
-- nb d'occurrences
nbocc :: (Eq a) => a -> [a] -> Int
nbocc _ [] = 0
nbocc x (y:ys) = (nbocc x ys) +
    (if x == y then 1 else 0)
```

```
> :t elem
???
```