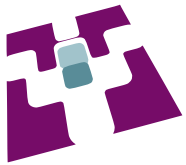
UNIVERSITÉ
CAEN
NORMANDIE

Autre Paradigme

Patrice Boizumault

Université de Caen - Normandie

Lundi 20 janvier 2020

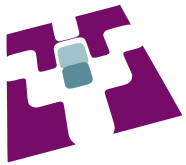


(rappel)

- Une liste est
  - soit vide `[]`
  - soit obtenue en ajoutant un élément à une liste, grâce à l'opérateur de **cons**truction `(:)`

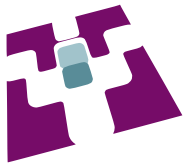
Exemple

```
[1, 7, 2, 5]
= 1 : [7, 2, 5]
= 1 : (7 : [2, 5])
= 1 : (7 : (2 : [5]))
= 1 : (7 : (2 : (5 : [])))
```



(rappel)

- En `Haskell`, toute liste est typée.
- Tous les éléments d'une liste doivent être de même type.
- **Exemples**
  - `[1, 7, 2, 5]` est de type `[Int]`
  - `"abc'e7q"` est de type `[Char]` (ou bien de type `String`)
  - `[True, False, True, True]` est de type `[Bool]`
  - `[True, 7, False]` est une liste mal typée $\rightarrow$ message d'erreur



(rappel)

- Primitives d'accès *head*, *tail* et de construction *(:)*

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

head :: [a] -> a

tail :: [a] -> [a]
```

- **Exemples**

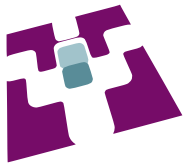
```
> head [1, 7, 2, 5]    ==> 1

> tail [1, 7, 2, 5]    ==> [7,2,5]

> head (tail [1, 7, 2, 5])    ==> 7

> tail (tail [1, 7, 2, 5])    ==> [2,5]

> head (tail (tail [1, 7, 2, 5]))    ==> 2
```

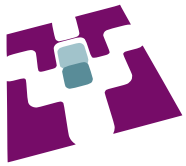


(rappel)

- Propriété : si `xs` est une liste d'éléments de même type que celui de `x`, alors on a toujours :
  - `head (x:xs) = x`
  - `tail (x:xs) = xs`
  - `(head xs):(tail xs) = xs`
- **Exemples**

```
head [7, 4, 8] = head (7 : [4, 8]) = 7
tail [7, 4, 8] = tail (7 : [4, 8]) = [4, 8]

head "hello" = head ('h' : "ello") = 'h'
tail "hello" = tail ('h' : "ello") = "ello"
```



(rappel)

- **Listes en compréhension (ZF-expressions)**

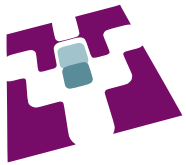
```
> [x | x <- [1..10]]    ==> [1,2,3,4,5,6,7,8,9,10]

> [x | x <- [1..10], even x]    ==> [2,4,6,8,10]

> [x * x | x <- [1..10], even x]    ==> [4,16,36,64,100]


> [(x,y) | x <- [1..2], y <- [5..7]]

        ==> [(1,5),(1,6),(1,7), (2,5),(2,6),(2,7)]
```

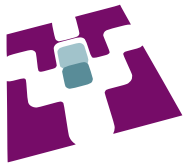


Pattern matching

- `[]` est le pattern (modèle) de la liste vide
- `(x:xs)` est le pattern (modèle) des listes non vides

Exemples

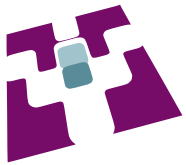
- `[1,3,2]` vs `(x:xs)` $\rightarrow$ `x=1`, `xs=[3,2]` car `[1,3,2]=1:[3,2]`
- `[3]` vs `(x:xs)` $\rightarrow$ `x=3`, `xs=[]` car `[3]=3:[]`
- `[1,3,2]` vs `(x:y:xs)` $\rightarrow$ `x=1`, `y=3`, `xs=[2]` car `[1,3,2]=1:3:[2]`
- `[1,1,2]` vs `(x:y:xs)` $\rightarrow$ `x=1`, `y=1`, `xs=[2]` car `[1,1,2]=1:1:[2]`
- `[3]` ne satisfait pas le pattern `(x:y:xs)`



Pattern matching

- `[]` est le pattern de la liste vide
- `(x:xs)` est le pattern des listes non vides
- `(x:xs)` est le pattern des listes ayant au moins un élément
- `(x:y:xs)` est le pattern des listes ayant au moins deux éléments

- Pour déterminer si une liste possède 2 premiers éléments identiques, peut-on utiliser le pattern `(x:x:xs)` ?
- Quelles listes représente le pattern `[x]` ?
- Quelles listes représente le pattern `[x, y]` ?



Traitement récursif de listes

Somme d'une liste d'entiers

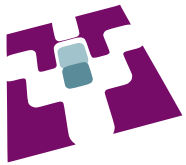
- type de cette fonction : `[Int] -> Int`
- deux définitions équivalentes :

```
sum1 :: [Int] -> Int
sum1 l = if (l == [])
         then 0
         else (head l) + (sum1 (tail l))

-- pattern matching

sum3 :: [Int] -> Int

sum3 [] = 0
sum3 (x:xs) = x + (sum3 xs)
```



Traitement récursif de listes

Longueur d'une liste

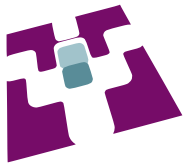
- type : `[a] -> Int`
- deux définitions équivalentes :

```
lg1 liste = if (liste == [])
            then 0
            else 1 + lg1 (tail liste)

lg3 [] = 0
lg3 (x:xs) = 1 + (lg3 xs)
```

- la variable `x` n'apparaît pas dans la partie droite de la 2nde équation, on peut remplacer `x` par l'attrape-tout noté `_`

```
lg3 (_:xs) = 1 + (lg3 xs)
```



Traitement récursif de listes

Concaténation de deux listes : opérateur infixe (++)

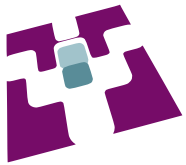
- exemples

```
append [] [1,2] = [] ++ [1,2]   ==> [1,2]
append [1,2,3] [4,5,6] = [1,2,3] ++ [4,5,6]  ==> [1,2,3,4,5,6]
```

- définition

```
append :: [a] -> [a] -> [a]

append []     ys = ys
append (x:xs) ys = x : (append xs ys)
```



(retour) sur le traitement récursif de listes

n-ième[1] élément d'une liste : opérateur infixe `(!!)`

- exemples

```
nEme 0 [20,30,40] = [20,30,40] !! 0  ==> 20
nEme 1 "abcd" = "abcd" !! 1  ==> 'b'

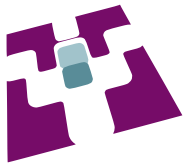
> "abcd" !! 1   ==> 'b'
> [20, 30, 40] !! 0   ==> 20
```

- définition

```
nEme :: Int -> [a] -> a

nEme 0 (x:_)  = x
nEme n (_:xs) = nEme (n-1) xs
```

[1] *La numérotation commence en 0*



Traitement récursif de listes

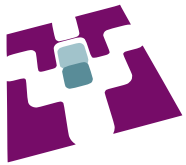
Retourner une liste

- exemples

```
reverse [1,2,3,4,5]  ==> [5,4,3,2,1]
reverse "qwertyuiop"  ==> "poiuytrewq"
```

- définition par 2 équations e_1 et e_2

```
reverse :: [a] -> [a]
reverse []     = []                          (e1)
reverse (x:xs) = (reverse xs) ++ [x]         (e2)
```



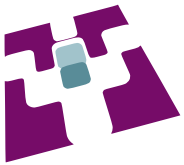
Traitement récursif de listes

Retourner une liste

- 2nde définition avec 3 équations e_1, e_2 et e_3

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

reverse xs = aux xs []                          (e1)
   where aux [] xs      = xs                    (e2)
         aux (x:xs) ys = aux xs (x:ys)           (e3)
```



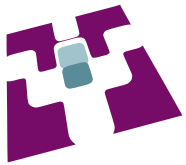
Traitement récursif de listes

Retourner une liste

- ```
  reverse :: [a] -> [a]
  reverse []     = []                          (e1)
  reverse (x:xs) = (reverse xs) ++ [x]         (e2)
  ```

- exemple de réduction d'une expression

```
reverse [1, 2, 3]
  = (reverse [2, 3]) ++ [1]                        (par e2)
  = (reverse [3] ++ [2])  ++  [1]                  (par e2)
  = ( (reverse [] ++ [3])  ++ [2])  ++ [1]         (par e2)
  = ( ([] ++ [3])  ++ [2])  ++ [1]                 (par e1)
  = [3, 2, 1]
```
```



Traitement récursif de listes

Dernier élément d'une liste

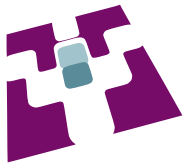
- polymorphisme de cette fonction

```
> last [3..12]  ==> 12
> last "hello"  ==> 'o'
> last [[2,5], [1,-3,-12], [7..10]]  ==> [7,8,9,10]
> last ["un", "deux", "trois"]  ==> "trois"
```

- type : `[a] -> a`
- définition

```
last [x] = x

last (_:x':xs) = last (x':xs)
```



Traitement récursif de listes

Une liste sauf son dernier élément

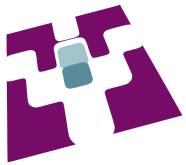
- polymorphisme

```
> init [7..10]   ==> [7,8,9]
> init "azerty"   ==> "azert"
> init [[2,5],[1,-3],[7..10]]  ==> [[2,5],[1,-3]]
> init  ["un", "deux", "trois"]  ==> ["un","deux"]
```

- type : `[a] -> [a]`
- définition

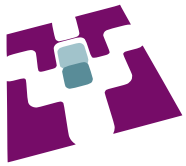
```
init [_] = []

init (x:xs) = x : (init xs)
```



Fonctionnelles

1. Fonctions anonymes
2. Multiples applications d'une fonction
3. Filtrer les éléments satisfaisant une propriété



Fonctionnelles

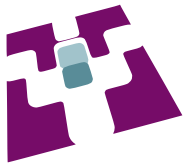
Fonctions anonymes (λ-abstraction)

- La notation $\lambda x.f(x)$ désigne la fonction : $x \rightarrow f(x)$
  - successeur sur les entiers : $\lambda x.(x+1)$
  - multiplier un nombre par 2 : $\lambda x.(2.x)$
  - être un entier strictement positif : $\lambda x.(x > 0)$
- La notation `Haskell`

```
\x -> f x
\x -> x + 1
\x -> x * 2
\x -> x > 0
```

- Utilisation

```
> (\x -> x + 1) 2    ==> 3
> (\x -> x * 2) (7+5)    ==> 24
> (\x -> x > 0) (head [1, 2, 7])    ==> True
```



Fonctionnelles

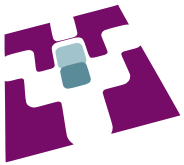
Fonctions anonymes (λ-abstraction)

- Fonctions avec plusieurs paramètres

```
\x y -> [x, y]          liste avec 2 e'l'ements
\x y -> (x+y)/2         moyenne de 2 nombres
\x y -> (x, y)          couple
```

- Utilisation

```
> (\x y -> [x, y] ) 6 85    ==> [6,85]
> (\x y -> [x, y] ) 'a' 'b'    ==> "ab"
> (\x y -> (x+y)/2) 7 14    ==> 10.5
> (\x y -> (x, y)) 7 12    ==> (7,12)
> (\x y -> (x, y)) 'a' 27    ==> ('a',27)
> (\x y -> (x, y)) "azerty" True    ==> ("azerty",True)
```



Notation préfixée/infixée des opérateurs binaires

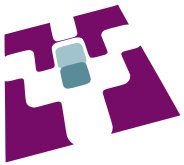
- notation infixée par défaut

```
> (7 * (8 + 2))      ==> 70
> 'a' : "zertyop"      ==> "azertyop
```

- utilisation en préfixé (en "parenthèsant" l'opérateur)

```
> (*) 7 (8 + 2)       ==> 70
> (*) 7 ((+) 8 2)      ==> 70
> (:) 'a' "zertyop"      ==> "azertyop"

> "ab" ++ "1zaw"      ==> "ab1zaw"
> (++) "ab" "1zaw"      ==> "ab1zaw"
```



Fonctionnelles

Application partielle[2] : opérateur (+)

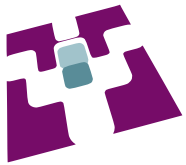
- `(+)` opérateur binaire de somme
- `(+ 2)` fonction qui ajoute 2 à un nombre
- `(+ 2)` et `(\x -> x+2)` sont 2 fonctions égales

```
> 2 + 7      ==> 9
> (+ 2) 7      ==> 9
> (\x -> x+2) 7      ==> 9
```

- les différents types (parenthèsage implicite à droite)

```
(+) :: Int -> (Int -> Int)
(+ 2) :: Int -> Int
(+ 2) 7 :: Int
```

[2] la notion de curryfication (cas général) sera présentée à la séance suivante.



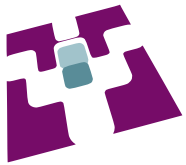
Application partielle : opérateur (==)

- `(==)` opérateur binaire
- `(== 'a')` fonction qui détermine si un caractère vaut `'a'`
- `(== 'a')` et `(\x -> x=='a')` sont 2 fonctions égales

```
> 'b' == 'a'      ==> False
> (== 'a') 'b'      ==> False
> (\x -> x=='a') 'b'      ==> False
```

- les différents types

```
(==) :: Char -> Char -> Bool
(== 'a') :: Char -> Bool
(== 'a') 'b' :: Bool
```



Multiples applications d'une fonction

- `(map f xs)` construit la liste des applications de la fonction `f` à tous les éléments de la liste `xs`

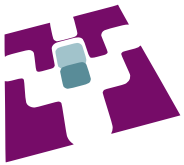
```
> map square [1..4]   ==> [1,4,9,16]
> map odd [1..4]    ==> [True, False, True, False]
> map (* 2) [1..4]    ==> [2, 4, 6, 8]
> map even [1..4]    ==> [False, True, False, True]
```

- typage

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

- somme des carrés des n premiers entiers

```
sumsquare :: Int -> Int
sumsquare n = sum (map square [1..n])
```



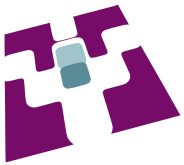
Multiples applications d'une fonction

- définition récursive de la fonction `map`

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

- autre définition à l'aide d'une ZF-expression

```
map f xs = [f x | x <- xs]
```



Fonctionnelles

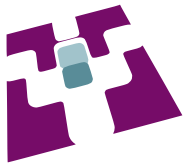
Multiples applications d'une fonction

- distributivité de `map` par rapport à la composition de fonctions

```
map (f.g) = (map f) . (map g)
```

- distributivité de `(map f)` par rapport à la concaténation

```
map f (xs ++ ys) = (map f xs) ++ (map f ys)
```



Fonctionnelles

Filtrer les éléments d'une liste satisfaisant une propriété

- `(filter p xs)` construit la liste des éléments de la liste `xs` satisfaisant le prédicat `p`

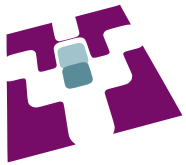
```
> filter even [1..10]   ==>  [2,4,6,8,10]
> filter odd [1..10]   ==> [1,3,5,7,9]
> filter (== 'a') "azertayaap"   ==> "aaaa"
> filter (/= 'a') "azertayaap"   ==> "zertyp"
```

- typage

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

- somme des carrés des nombres pairs entre `1` et `n`

```
sumEvenSquare :: Int -> Int
sumEvenSquare n = sum (map square (filter even [1..n]))
```



Fonctionnelles

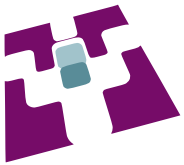
Filtrer les éléments d'une liste satisfaisant une propriété

- définition récursive de `filter`

```
filter :: (a -> Bool) -> [a] -> [a]
filter p []    = []
filter p (x:xs)
   | (p x)     = x:(filter p xs)
   | otherwise = filter p xs
```

- autre définition à l'aide d'une ZF-expression

```
filter p xs = [x | x <- xs, p x]
```



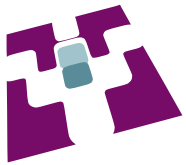
Filtrer les éléments d'une liste satisfaisant une propriété

- commutativité par rapport à la composition de fonctions

```
(filter p) . (filter q) = (filter q) . (filter p)
```

- distributivité de `(filter p)` sur `(++)`

```
filter p (xs ++ ys) = (filter p xs) ++ (filter p ys)
```



Retour sur les fonctions anonymes (λ-abstraction)

- Eviter de définir des fonctions de peu d'intérêt général ou bien utilisées ponctuellement

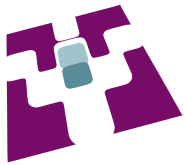
```
> map (\x->x+1) [1..10]  ==> [2,3,4,5,6,7,8,9,10,11]

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

- Les fonctions anonymes peuvent utiliser le filtrage

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

-- les moins de 25 ans
> filter (\(nom,age) -> (age <= 25)) xs
[("Jean",23),("Marie",25),("Anne",18)]
```



Retour sur les fonctions anonymes (λ-abstraction)

- Les fonctions anonymes peuvent utiliser le filtrage (suite)

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

-- les moins de 25 ans
> filter (\(_, age) -> (age <= 25)) xs
[("Jean",23),("Marie",25),("Anne",18)]

-- Les couples d'entiers ordonnés
> filter (\(x,y) -> (x<y)) [(1,4), (5,2), (2, 9)]
[(1,4), (2,9)]

lesordonnes :: [(Int, Int)]  ->  [(Int, Int)]
lesordonnes xs = filter (\(x,y) -> (x<y)) xs
```