



Programmation fonctionnelle

Master Informatique
Module I77UD05

Thierry Lemeunier
thierry.lemeunier@univ-lemans.fr

Présentation du module (1/2)

- Objectifs pédagogiques
 - Introduction au paradigme de la programmation fonctionnelle
≠ programmation impérative
≠ programmation orientée-objet
≠ programmation logique
 - Introduction au langage Haskell
 - Nous ne verrons pas l'intégralité du langage mais suffisamment pour montrer l'intérêt du langage et faire des TP intéressants
- Organisation
 - Intervenant : Thierry Lemeunier
 - Volume horaire : 4 HC + 20 HTP
 - Evaluation :
 - Examen écrit + note de TP
 - 7 TP = 3 séances de « TD sur machine » + 4 séances de TP notées
 - 1^{ère} session :
$$(2,25 * \text{Note_Ecrit1} + 0,75 * \text{Note_TP}) / 3$$
 - 2^{ième} session :
$$(2,25 * \text{Note_Ecrit2} + 0,75 * \text{Report_Note_TP}) / 3$$

Présentation du module (2/2)

- Références bibliographiques
 - « Programming in Haskell », G. Hutton, Cambridge University Press, 978-1-316-62622-1
 - « Apprendre Haskell vous fera le plus grand bien ! », Miran Lipovača, <http://haskell.fr/lyah/>
 - Site web officiel : <https://www.haskell.org>
 - Documentation des modules officiels Haskell <https://downloads.haskell.org/~ghc/latest/docs/html/libraries/index.html>
 - « Real World Haskell », B. O'Sullivan, J. Goerzen et D. Stewart, O'REILLY, 978-0-596-51498-3

Programmation fonctionnelle ?

- Paradigme basé sur la théorie mathématique du *lambda calcul* (Church, 1930) et de ses extensions
- Dans le paradigme fonctionnel, les traitements de données ≈ *l'application* de fonctions mathématiques
 - Fonction mathématique :
 - « Boîte noire » dont le résultat dépend uniquement des paramètres
 - Le résultat est toujours identique si les paramètres sont identiques
 - Une fonction n'a pas d'effet de bord = pas de modification / dépendance extérieure
- Une programmation sans **effet de bord**
 - Programmes plus sûr (pas de données partagées = pas de concurrence)
 - Fonction sans effet de bord = **fonction pure**
 - **Fonction impure** = fonction dépendante du « monde extérieur » au programme (fichiers, flux réseaux, etc.)
- Exécution par **transparence référentielle** (= par substitution) :
 - Le résultat ne change pas si on remplace une expression par une autre expression si c'est deux expressions sont égales mathématiquement
 - L'appel d'une fonction pure (= son application) peut être substitué par le résultat de son appel (= sa valeur de substitution)

Caractéristiques d'Haskell



- Langage issu du monde universitaire <https://www.haskell.org>
 - Travaux de recherche débutant en 1987
 - Première version stable en 2003 ; mise à jour importante en 2010
- Langage typé statiquement
 - **Contrôle strict des types** à la compilation
 - Inférence de type (quand c'est possible)
 - Définition de **structures de données** (récursives si nécessaire)
 - Puissance du système des classes interfaces en POO appliquée aux types avec les **classes de types**
- Langage fonctionnel récursif
 - **Fonction d'ordre supérieur** : fonction qui prend une fonction comme argument ou qui retourne une fonction comme évaluation
 - Séparation nette entre fonction pure et fonction impure
 - **Polymorphisme** : types et fonctions génériques paramétrés
- **Evaluation paresseuse**
 - L'évaluation est repoussée jusqu'au moment nécessaire
 - Possibilité de définir des listes infinies...
- Utilisations ? A tous domaines de l'industrie informatique
 - Programmation système / concurrente / multi-cœur / réseau / mathématique / base de données...
 - Interfaçage graphique possible (librairie gtk2hs)
 - Programmation modulaire (différents espace de nom) ; gestionnaire de package...
 - Compilateur : GHC ; Interpréteur en ligne de commande : GHCI

Mes premières fonctions (1/3)

- Une fonction Haskell prend la forme d'une « équation mathématique »
- Une fonction est définie par :
 - un nom commençant par une minuscule ;
 - un ou plusieurs arguments nommés (sans parenthèse) ;
 - un corps qui est la valeur de substitution de la fonction ;
 - et un type déduit ou explicite (cf. ci-après)
- L'exécution d'une fonction consiste à **appliquer** la fonction jusqu'à obtenir une valeur finale (\approx dont l'évaluation est elle-même)

```
double x = x + x -- Fonction nommée double à un argument x
```

Commentaire
commençant par –

Evaluation « interne » d'abord ou **évaluation par valeur**

```
double (double 2)  
⇒ double (2 + 2)  
⇒ double 4  
⇒ 4 + 4  
⇒ 8
```

En programmation
fonctionnelle, l'ordre
d'évaluation n'a pas
d'importance sur le résultat

Evaluation « externe » d'abord ou **évaluation pas nom**

```
double (double 2)  
⇒ (double 2) + (double 2)  
⇒ (2 + 2) + (double 2)  
⇒ 4 + (double 2)  
⇒ 4 + (2 + 2)  
⇒ 4 + 4  
⇒ 8
```

Par défaut, Haskell utilise l'évaluation par nom
(avec une optimisation pour éviter les évaluations redondantes)

Mes premières fonctions (2/3)

- En Haskell, l'**évaluation est paresseuse** : les expressions sont évaluées uniquement quand c'est nécessaire

Caractère underscore
Haskell ≈
caractère
étoile du
shell Unix

ones :: [Int]

ones = 1 : ones

-- Type de la fonction : retourne une liste d'entier

-- Fonction récursive nommée ones sans argument

head :: [a] -> a

head (x : _) = x

-- Prend une liste contenant un type *a* et retourne un type *a*

-- Retourne la tête de la liste (le reste de la liste n'importe pas)

Evaluation pas nom

ones

⇒ 1 : ones

⇒ 1 : 1 : ones

⇒ 1 : 1 : 1 : ones

⇒ 1 : 1 : 1 : 1 : ones

⇒ etc.

Haskell peut manipuler
des **structures infinies** !

Evaluation pas valeur

head ones

⇒ application de ones

head (1 : ones)

⇒ application de ones

head (1 : 1 : ones)

⇒ application de ones

head (1 : 1 : 1 : ones)

⇒ etc.

Evaluation pas nom

head ones

⇒ application de ones

head (1 : ones)

⇒ application de head

1

L'évaluation s'arrête
dès que l'expression
correspond au membre
gauche de l'équation

Rappel : par défaut, Haskell
utilise l'évaluation par nom

Mes premières fonctions (3/3)

- Syntaxe d'une fonction Haskell

Mathématique	Haskell
$f(x)$	$f\ x$
$f(x, y)$	$f\ x\ y$
$f(g(x))$	$f\ (g\ x)$
$f(x, g(y))$	$f\ x\ (g\ y)$

- Exemples

Les fonctions ont priori maximale

Les fonctions sur les listes sont souvent récursives...

- Convention standard des priorités

fonction > ^	>	*	/	>	+	-
association à droite				association à gauche		
$f\ a + b \Leftrightarrow (f\ a) + b$				$2 - 3 + 4 \Leftrightarrow (2 - 3) + 4$		
$2^3^4 \Leftrightarrow 2^3(3^4)$				-- Parenthèses inutiles		
sqrt 2 + sqrt 3				-- Parenthèses nécessaires		
compare (sqrt 2) (sqrt 3)						

sum :: Num a => [a] -> a -- Type de la fonction : prend une liste de type a (contrainte sur a)
sum [] = 0 -- 1^{ère} équation : retourne 0 si appliquée sur une liste vide
sum (x : xs) = x + sum xs -- 2^{ème} équation : retourne 1 + résultat sur le reste de la liste

qsort :: Ord a => [a] -> [a] -- Prend une liste de type a et retourne une liste de type a
qsort [] = [] -- 1^{ère} équation : retourne liste vide si appliquée sur une liste vide
-- 2^{ème} équation : retourne une liste triée
qsort (x : xs) = qsort smaller ++ [x] ++ qsort larger
where smaller = [a | a <- xs, a <= x]
larger = [a | a <- xs, a > x]

Cf. le type liste et les définitions locales

En Haskell l'indentation est nécessaire.
Préférez les espaces aux tabulations.

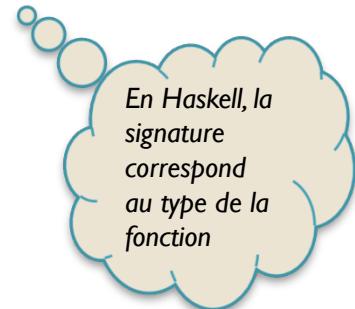
Les types prédéfinis (1/2)

- En Haskell toute expression est typée (fonction, valeur, etc.)
- Un type est un ensemble de valeurs possibles (\approx un ensemble mathématique)
En principe $\text{Bool} = \{ \text{True}, \text{False} \}$
- L'opérateur '`::`' s'applique à toute expression
 $e :: T \Leftrightarrow$ l'expression e est de type T
- Une fonction possède un type puisqu'elle retourne une valeur ; le type d'une fonction tient aussi compte du type des arguments

not :: Bool -> Bool -- Type de la fonction *not* : prend un Bool et retourne un Bool

- Le type retourné par l'application d'une fonction est défini par son type
- Les types prédéfinis
 - Les booléens : **Bool** = { True, False }
 - Les caractères : **Char** = { 'c' | 'c' \in Tables_Unicode }
 - Les chaînes : **String** = { "string" | $\forall c \in$ "string" , $c \in$ Char }
 - Les « petits entiers » : **Int** = { entier | $-2^{63} \leq$ entier $\leq 2^{63}-1$ }
 - Les « grands entiers » : **Integer** = { entier | taille(entier) limitée par la mémoire }
 - Les « petits décimaux » : **Float** = ensemble des décimaux en simple précision
 - Les « grands décimaux » : **Double** = ensemble des décimaux en double précision

f :: A -> B	e :: A
f e :: B	



En Haskell, la signature correspond au type de la fonction

Les types prédéfinis (2/2)

- Les types prédéfinis (suite)
 - Les listes : **List** = { e | $\forall e \in \text{type } T$ }
 - Remarque 1 : le type String est en fait une liste de caractères => String \Leftrightarrow [Char]
 - Remarque 2 : [a] \Leftrightarrow [] a

[T] \Leftrightarrow Liste de type T
[] \Leftrightarrow Liste vide de longueur 0
[[]] \neq []

[False, True, True] :: [Bool]
['a', 'b', 'c'] :: [Char]
["toto", "titi"] :: [String]
[['a', 'b', 'c'], ['d', 'e']] :: [[Char]]

Liste définie par progression

[1..5] \Leftrightarrow [1, 2, 3, 4, 5]
['a'..'z'] \Leftrightarrow 26 minuscules de l'alphabet
[1,4..10] \Leftrightarrow [1,4,7,10] liste par pas de 3
[10,9..1] \Leftrightarrow liste par ordre décroissant
[1..] \Leftrightarrow liste infinie

Liste définie par compréhension

[2 * x | x <- [1..10]] -- liste des 10 premiers nombres pairs
[2 * x | x <- [1..10] , 2*x <= 12] -- idem précédent mais < 12
[(x, y) | x <- [1, 2] , y <- [4, 5]] \Leftrightarrow [(1,4), (1,5), (2,4), (2, 5)]
length xs = sum [1 | _ <- xs] -- longueur de la liste xs

Quel triangle rectangle a des côtés entiers tous inférieurs ou égaux à 10 et a un périmètre de 24 ?

[(a, b, c) | c <- [1..10] , b <- [1..10] , a <- [1..10] ,
 $a^2+b^2 == c^2$, $a+b+c == 24$]

- Le type **tuple** : une séquence limitée d'éléments de types éventuellement différents

Tuple d'arité 0 ou tuple vide : ()
 Tuple d'arité 2 ou une paire : (a, b)
 Tuple d'arité 3 ou une triplette : (a, b, c)
 Etc.

(False, 'a', 10) :: (Bool, Char, Int)
(‘a’, (False, 10)) :: (Char, (Bool, Int))
fst (1, 2) -- Premier élément d'une paire
snd (1, 2) -- Second élément d'une paire

Fonctions simples sur les listes

- Construction avec l'opérateur `cons` ':'
- Concaténation avec l'opérateur '++'
- Longueur d'une liste : `length :: [a] -> Int`
- Teste si la liste est vide : `null :: [a] -> Bool`
- Tête d'une liste : `head :: [a] -> a`
- Dernier élément : `last :: [a] -> a`
- Reste d'une liste : `tail :: [a] -> [a]`
- Liste sans le dernier élément : `init :: [a] -> [a]`
- Les n premiers éléments : `take :: Int -> [a] -> [a]`
- Retirer les n premiers éléments : `drop :: Int -> [a] -> [a]`

`1 : [4, 5] ⇔ [1, 4, 5]`
`1 : 2 : 3 : [] ⇔ 1 : (2 : (3 : [])) ⇔ [1, 2, 3]`
`'a' : "bcd" ⇔ "abcd"`

`[3, 4, 5] ++ [6, 7] ⇔ [3, 4, 5, 6, 7]`
`"abc" ++ "toto" ⇔ "abctoto"`

`length [1, 2, 3] ⇔ 3`
`length [] ⇔ 0`
`null [1,2,3] ⇔ False`
`null [] ⇔ True`

`head [1,2,3] ⇔ 1`
`last "toto" ⇔ 'o'`
`tail [4, 5, 6] ⇔ [5, 6]`
`init [7, 8, 9] ⇔ [7, 8]`

`take 2 ['a', 'b', 'c', 'd'] ⇔ ['a', 'b']`
`drop 2 ['a', 'b', 'c', 'd'] ⇔ ['c', 'd']`

Les types des fonctions (1/2)

- Le type d'une fonction indique le type des paramètres et le type retourné

```
f0 :: T      -- Fonction f0 sans argument retourne le type T
f1 :: T1 -> T2      -- T1 : le type du 1er paramètre
                      -- T2 : le type retourné
f2 :: T1 -> T2 -> T3 -- T1 et T2 : deux paramètres
                      -- T3 : le type retourné
```

```
add :: (Int, Int) -> Int
add (n, m) = n + m

zeroto :: Int -> [Int]
zeroto n = [0..n]
```

```
take :: Int -> [a] -> [a]          -- Prend un entier puis prend une liste et
                                         -- retourne une liste de même type
take 0 _      = []                  -- 1ère équation : retourne liste vide si l'entier est zéro
take _ []     = []                  -- 2ème équation : retourne liste vide si la liste est vide
take n (x:xs) = x : take (n - 1) xs -- 3ème équation : les autres cas
```

- Une fonction peut être totale ou partielle
 - Fonction totale : quelque que soit les valeurs des arguments la fonction retourne une valeur
 - Fonction partielle : certaines valeur d'arguments lèvent une exception (par exemple head [])
- Opérateur d'application \$ d'une fonction *f* unaire : $f x \Leftrightarrow f \$ x$

```
head (drop 4 ("abcdef" ++ "123"))    ⇔    head $ drop 4 ("abcdef" ++ "123")
```

- Fonction binaire utilisée en notation infixe

```
plus 1 2    ⇔    1 `plus` 2
```

Les types des fonctions (2/2)

- Les fonctions polymorphes (= fonctions génériques paramétrées)
 - Ce sont des fonctions ayant pour argument ou valeur rentrée un type indéterminé
 - Les types de ces fonctions utilisent des variables de type (symbolisés par des noms en minuscule) : les types de ces fonctions sont donc paramétrés
 - Si le type est polymorphe alors la fonction est polymorphe

<code>fst :: (a, b) -> a</code>	-- Prend une paire quelconque et retourne le premier composant -- Le type <i>a</i> peut être différent du type <i>b</i> ou le même
<code>take :: Int -> [a] -> [a]</code>	-- Prend un entier et une liste de type <i>a</i> quelconque -- et retourne une liste de même type <i>a</i>
<code>head :: [a] -> a</code>	-- Prend une liste de type <i>a</i> quelconque -- et retourne une valeur de type <i>a</i>
<code>zip :: [a] -> [b] -> [(a, b)]</code>	-- Prend deux listes de types <i>a</i> et <i>b</i> quelconques -- et retourne une liste de paire de type <i>(a, b)</i>

> signifie que
l'on est dans
l'interpréteur
Haskell

```
> fst (1, ["toto", "titi"])
1
> fst ("toto", "titi"), 1
["toto", "titi"]
```

```
> head [1,2,3]
1
> head "toto"
't'
```

```
> zip "toto" [1,2,3]
[('t', 1), ('o', 2), ('t', 3)]
> head $ zip "toto" [1,2,3]
('t', 1)
```

Evaluations conditionnelles

- Expression conditionnelle *if...then...else...*
 - Les deux « branches » doivent retourner le même type !
 - On peut emboîter plusieurs expressions conditionnelles

```
signum :: Int -> Int
signum n = if n < 0
           then -1
           else if n == 0
                 then 0
                 else 1
```

- Equation gardée :
 - C'est une équation suivie d'une séquence de conditions logiques
 - Une équation gardée est généralement plus lisible

```
signum :: Int -> Int
signum n | n < 0          = -1
         | n == 0          = 0
         | otherwise        = 1    -- otherwise == True
```

Les variables en Haskell ?

- L'expression *let...in* permet des définitions locales à une fonction
- *let* peut masquer des paramètres ou d'autres définitions locales

```
estRectangle :: Int -> Int -> Int -> Bool
estRectangle a b c =
    let res1 = a^2 + b^2 == c^2
        res2 = a^2 + c^2 == b^2
        res3 = b^2 + c^2 == a^2
    in res1 || res2 || res3
```

```
test a = let a = [1,2,3]    -- La définition a masque
          in a ++ [4,5,6]  -- l'argument a
```

- L'expression *where*
placée en fin de fonction
permet d'ajouter des définitions locales

```
lend amount balance
| amount <= 0 = Nothing
| amount > reserve * 0.5 = Nothing
| otherwise = Just newBalance
where reserve = 100
      newBalance = balance - amount
```

Nothing et Just
sont du type Maybe

Déclaration de types (1/3)

- Type synonyme avec l'expression **type** (= facilite la lecture du code)
 - Un nouveau type peut être défini comme synonyme d'un type déjà existant

```
type String = [Char]      -- 1er caractère du nom en majuscule  
type Pos = (Int, Int)    -- Une position  
type Trans = Pos -> Pos -- Type d'une fonction
```

- Un type synonyme ne peut être défini récursivement !
- Un type synonyme peut être polymorphe
- Un type synonyme peut avoir plusieurs variables de type

```
type Paire a = (a, a)
```

```
type Assoc k v = [(k, v)] -- Liste (clé, valeur)
```

```
find :: Eq k => k -> Assoc k v -> v  
find k ts = head [v' | (k', v') <- ts, k' == k]
```

- Type de données algébriques avec l'expression **data**

- Une façon de définir un type de données est de définir l'ensemble des valeurs possibles

```
data Bool = False | True
```

-- Le type *Bool* ne comprend que deux valeurs
-- 1^{ère} lettre en majuscule pour tous les constructeurs

Un constructeur de type

Un ou plusieurs constructeurs de valeur

Déclaration de types (2/3)

- Un type synonyme ou un type algébrique peut être utilisé comme n'importe quel autre type notamment avec les fonctions

```
type Pos = (Int, Int)
data Move = North | South | East | West
```

```
move :: Move -> Pos -> Pos
move North (x, y) = (x, y + 1)
move South (x, y) = (x, y - 1)
...
```

- Un constructeur de valeur peut avoir des arguments
- Un constructeur de valeur est une fonction retournant le nouveau type (et réciproquement : une fonction peut être utilisée comme un constructeur de valeur)

```
data Shape = Circle Float | Rect Float Float
square :: Float -> Shape
square n = Rect n n
area :: Shape -> Float
area (Circle r) = pi * r ^ 2
area (Rect x y) = x * y
```

```
> :type Circle
Circle :: Float -> Shape

> :type Rect
Rect :: Float -> Float -> Shape

> :type Rect 10 4
Rect 10 4 :: Shape
```

Déclaration de types (3/3)

- Un type de données peut être **polymorphe** (c'est-à-dire paramétrisé)
- Un type de données peut être défini récursivement

```
data BinaryTree a = Leaf a | Node (BinaryTree a) a (BinaryTree a)
```

```
flatten :: BinaryTree a -> [a]
```

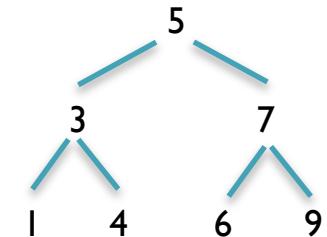
```
flatten (Leaf x) = [x]
```

```
flatten (Node left x right) = flatten left ++ [x] ++ flatten right
```

```
> atree = Node (Node (Leaf 1) 3 (Leaf 4)) 5 (Node (Leaf 6) 7 (Leaf 9))
```

```
> flatten atree
```

```
[1,3,4,5,6,7,9]
```



- Type simple avec l'expression **newtype**

- Un type synonyme (défini avec *type*) n'est pas vraiment un nouveau type

```
type Nat = Int           => Nat et Int sont synonymes : ils sont interchangeables
```

```
newtype Nat = N Int      => Nat et Int seront deux types différents
```

```
data Nat = N int          => newtype est géré plus efficacement que data
```

Nouveau type

- Un type *data* ou *newtype* peut instancier une classe de type (un type synonyme ne peut pas)
- *newtype* est utilisable uniquement avec **un seul constructeur de valeur à un seul argument**
- Comme *data* et *type*, *newtype* peut être polymorphe

```
newtype Parser a = P (String -> [(a, String)]) -- Un type qui encapsule une fonction polymorphe
```

Les lambdas expressions



- Une lambda expression permet de définir une fonction anonyme

```
\x -> x + x -- lambda expression à un argument nommé x
```

- Une lambda expression est utilisée comme toute fonction

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

```
> (\x y -> x + y) 2 6  
8
```

Une lambda expression commence par \ ce qui évoque λ

- Une lambda expression peut être utilisée lorsqu'une fonction retourne une fonction :

```
myadd :: a -> (a -> a)  
myadd x = \y -> x + y
```

```
> :type (myadd 1)  
(myadd 1) :: Num a => a -> a  
> (myadd 1) 2  
3
```

- Une lambda expression peut définir une fonction utilisée localement :

```
odds :: Int -> [Int]  
odds n = map (\x -> x * 2 + 1) [0..n-1]
```

La fonction map applique une fonction à chaque élément d'une liste et retourne la liste des résultats

Application partielle (1/2)

- Toute fonction à plusieurs arguments peut être traitée comme une fonction à un seul argument retournant une fonction

Fonction à deux arguments

```
myplus :: Int -> Int -> Int  
myplus x y = x + y
```



```
myplus :: Int -> ( Int -> Int )  
myplus x = \y -> x + y
```

Fonction à un argument

- Autrement dit, la fonction *myplus* binaire peut être appliquée comme une fonction unaire : c'est une **application partielle**
- De manière générale, toute fonction à n arguments est l'équivalent d'une fonction sans argument retournant une lambda expression

Fonction à zéro argument

```
myplus :: ( Int -> (Int -> Int) )  
myplus = \x -> (\y -> x + y)
```



```
myplus :: (Int -> Int -> Int)  
myplus = \x y -> x + y
```

Fonction à zéro argument

- L'application partielle consiste à « fixer » zéro, un ou plusieurs paramètres
- On dit que la fonction est *curryfiée* (d'après le nom de Haskell Curry)

```
> myplus 1 2    -- deux arg.  
3  
> (myplus 1) 2  -- un arg.  
3  
> (myplus) 1 2   -- zéro arg.  
3
```

```
> :type myplus 1 2  
myplus 1 2 :: Int  
> :type myplus 1  
myplus 1 :: Int -> Int  
> :type myplus  
myplus :: Int -> Int -> Int
```

Application partielle (2/2)

- Cela peut conduire à des expressions étranges :

La fonction elem teste si un élément fait partie d'une liste

```
> elem 'g' ['a'..'z']
True
> (elem 'g') ['a'..'z']
True
> 'g' `elem` ['a'..'z']
True
> (`elem` ['a'..'z']) 'g'
True
```

- Opérateur currifié

- Un opérateur est une fonction infixe binaire (+ ; - ; * ; / ; ++ ; \$; etc.)
- Pour un opérateur # quelconque les expressions (#) (x #) et (# y) sont définis ainsi :

```
(#) = \x -> (\y -> x # y)      -- Opérateur currifié retournant une lambda à un argument
```

```
(x #) = \y -> x # y          -- Opérateur currifié fixant l'argument de gauche
                                         -- retournant une lambda en attente de l'argument de droite
```

```
(# y) = \x -> x # y          -- Opérateur currifié fixant l'argument de droite
                                         -- retournant une lambda en attente de l'argument de gauche
```

```
("1"++) "2"  =>  "12"
(++ "1") "2"  =>  "21"
(++) "1" "2"  =>  "12"
```

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

Les classes de type (1/4)

- Une classe de type définit un « contrat » devant être honoré par une « instance » de la classe (\approx notion de classe interface en POO)

(+) :: Num a => a -> a -> a -- *Num a est une contrainte sur le type du paramètre a*

paire :: (Num a, Eq a) => a -> Bool -- *Num a, Eq a est une contrainte sur le type du paramètre a*

- Une classe de type est formalisée en déclarant les fonctions attendues
- Un type qui instancie une classe doit avoir toutes ces fonctions implémentées
- Il existe beaucoup de classes de type prédéfinis
- Le programmeur peut définir de nouvelles classes de type (cf. ci-après)

classe Eq : classe des types dont les valeurs sont comparables entre elles

Fonctions :

(==) :: a -> a -> Bool -- Test d'égalité
(/=) :: a -> a -> Bool -- Test de différence

Instances :

Bool, Char, String, Int, Integer, Float, Double
[] et tuples s'ils sont constitués de types instances de Eq

Comme
en POO

Les instances peuvent
utilisées les fonctions
de la classe

Les classes de type (2/4)

classe **Ord** : classe des types Eq dont les valeurs sont ordonnables entre elles

Fonctions :

(<) :: a -> a -> Bool
(<=) :: a -> a -> Bool
(>=) :: a -> a -> Bool
(>) :: a -> a -> Bool
min :: a -> a -> a
max :: a -> a -> a

-- Teste si le 1^{er} arg. est inférieur au 2^{ème} arg.
-- Teste si le 1^{er} arg. est inférieur ou égal au 2^{ème} arg.
-- Teste si le 1^{er} arg. est supérieur ou égal au 2^{ème} arg.
-- Teste si le 1^{er} arg. est supérieur au 2^{ème} arg.
-- Retourne le plus petit des deux arguments
-- Retourne le plus grand des deux arguments

Instances :

Bool, Char, String, Int, Integer, Float, Double

[] et tuples s'ils sont constitués de types instances de Ord

Exemples :

1 < 2 ⇔ True

min 2 3 ⇔ 3

('a', 2) < ('b', 1) ⇔ True

('a', 2) < ('a', 1) ⇔ False

"Elephant" < "elephant" ⇔ True

ATTENTION

Le premier article du premier tuple est comparé au premier article du deuxième tuple; s'ils ne sont pas égaux, c'est le résultat de la comparaison, sinon le deuxième article est considéré, puis le troisième et ainsi de suite.

Les majuscules sont avant les minuscules

Les classes de type (3/4)

classe **Show** : classe des types dont les valeurs sont convertibles en chaîne de caractère

Fonctions :

`show :: a -> String` -- Conversion du type *a* en type *String*

Instances :

Bool, Char, String, Int, Integer, Float, Double

[] et tuples s'ils sont constitués de types instances de Show

Exemples :

`show False` \Leftrightarrow "False"

`show [1,2,3]` \Leftrightarrow "[1,2,3]"

`show 123` \Leftrightarrow "123"

classe **Read** : classe des types dont les valeurs sont convertibles depuis une chaîne

Fonctions :

`read :: String -> a` -- Conversion du type *String* en type *a* si le contexte le permet

Instances :

Bool, Char, String, Int, Integer, Float, Double

[] et tuples s'ils sont constitués de types instances de Read

Exemple : `> read "False" :: Bool` -- L'indication du type attendu est nécessaire
False

L'inférence de type peut échouer avec read

Les classes de type (4/4)

classe Num : classe des types dont les valeurs sont numériques

Fonctions :

(+) :: a -> a -> a	-- Addition de deux nombres
(-) :: a -> a -> a	-- Soustraire le 2 ^{ème} nombre du 1 ^{er} nombre
(*) :: a -> a -> a	-- Multiplier les deux nombres
negate :: a -> a	-- Retourner l'opposé de l'argument
abs :: a -> a	-- Retourner la valeur absolue de l'argument
signum :: a -> a	-- Retourner le signe de l'argument (-1 ou 0 ou 1)

Instances :

Int, Integer, Float, Double

- D'autres classes de type existent :
 - Integral pour les nombres entiers (fonctions *div* et *mod*)
 - Fractional pour les nombres fractionnaires (fonctions */* et *recip*)
 - Enum pour les énumérations ordonnées (fonctions *succ*, *pred*, etc.)
 - Bounded pour des types à valeur bornées (fonctions *minBound* et *maxBound*)
 - Etc.
- D'autres classes de type sont abordées en fin de cours

Programmation modulaire

- Un programme Haskell est une collection de modules
- Un module principal charge les autres modules et les utilise
- Un module est un fichier source définissant une collection de types, classes et fonctions liés les uns aux autres et formant un tout cohérent
- Déclaration d'un module : en début de fichier utiliser l'expression *module*

```
module NomDuModule (liste, des, éléments, exportés) where  
...
```

```
module Main () where  
...  
main = ...
```

Le module principal contient la fonction main
Le module principal n'exporte jamais rien !

La fonction main
est exécutée
automatiquement

- Importation d'un module
 - Depuis l'interpréteur exécutez la commande `:m + NomDuModule`
 - Dans un fichier source, l'expression *import NomDuModule [(éléments, du, module, à, importer)]*
- Eviter les conflits de nom
 - 1^{ère} solution : masquer un élément importé
 - 2^{ème} solution : ajouter un préfixe aux éléments importés

```
import Prelude hiding (lookup)  
...
```

```
import qualified Data.Map as M  
...  
compterLettres = foldr (M.alter ...) ...
```

Pattern matching

(Correspondance de motif)

- Définition de fonctions par pattern matching sur les arguments

```
not :: Bool -> Bool  
not False = True  
not True = False
```

```
(&&) :: Bool -> Bool -> Bool  
True && b = b  
False && _ = False
```

- Pattern matching sur les tuples : un tuple de patterns est un pattern

```
fst :: (a, b) -> a  
fst (x, _) = x
```

```
snd :: (a, b) -> b  
snd (_, x) = x
```

- Pattern sur les listes : une liste de patterns est une liste

```
test3 :: [Char] -> Bool  
test3 ['a', _, _,_] = True  
test3 _ = False
```

```
testn :: [Char] -> Bool  
testn ('a' : _) = True  
testn _ = False
```

```
head :: [a] -> a  
head (x : _) = x
```

```
tail :: [a] -> [a]  
tail (_ : xs) = xs
```

- Pattern sur les constructeurs de valeur

```
data Shape = Circle Float Float Float | ...  
  
surface :: Shape -> Shape  
surface (Circle _ _ r) = pi * r^2  
surface ...
```

- Pattern sur une expression

```
type Parser a = String -> [(a, String)]  
  
item :: Parser Char  
item = \s -> case s of  
    []      -> []  
    (x:xs) -> [(x, xs)]
```

Gérer les fonctions partielles (1/2)

L'opérateur / gère la division par zéro

- Rappel : certaines fonctions sont partielles

```
> head []
```

*** Exception: Prelude.head: empty list

```
> 1 `div` 0
```

*** Exception: divide by zero

```
> 1 / 0
```

Infinity

```
> :type 1 / 0
```

1/0 :: Fractional a => a

- Lever une exception : la fonction `error :: String -> a`

```
-- Retourne le second élément d'une liste
mysnd :: [a] -> a
mysnd xs = if length xs < 2
            then error "List too short"
            else head (tail xs)
```

```
> mysnd [1, 2]
2
> mysnd [1]
*** Exception: List too short
CallStack (from HasCallStack):
    error, called at <interactive>:32:34 in interactive:Ghci13
```

- Gérer les cas d'erreurs : le type `Maybe`

- Le type `Maybe` permet de gérer les cas d'erreurs des fonctions partielles
- Il signifie en substance : le type retourné peut être du type `a` mais ce n'est pas certain

```
data Maybe a = Nothing | Just a
deriving (Eq, Ord)
```

Instanciation automatique (cf. ci-après)

```
mysnd :: [a] -> Maybe a
mysnd (_ : x : _) = Just x
mysnd _ = Nothing
```

Gérer les fonctions partielles (2/2)

- Gérer les cas d'erreurs : le type Either
 - Le type Either est un type ayant deux valeurs possibles Left ou Right
 - Par convention, le constructeur Left doit être retourné en cas d'erreur

```
data Either a b = Left a | Right b  
deriving (Eq, Ord, Read, Show)
```

```
import Data.Char ( digitToInt, isDigit )  
  
parseEither :: Char -> Either String Int  
parseEither c  
| isDigit c = Right (digitToInt c)  
| otherwise = Left "parse error"
```

Importation depuis un module

- Gérer les exceptions dans les fonctions impures
 - Module System.IO.Error
 - La fonction catchIOError :: IO a -> (IOError -> IO a) -> IO a

Cf. le type
IO a ci-après
dans le cours

Définition de classe de type (1/2)

- Rappel : une classe de type est l'équivalent de la classe interface de la POO
- Une classe de type est définie avec l'expression `class...where`
- Une classe de type est toujours polymorphe car elle s'applique à des types différents

Une classe de type peut fournir une implémentation par défaut

```
class Eq a where
  (==), (/=) :: a -> a -> Bool

  x /= y = not (x == y)
```

- Les instances d'une classe sont définies avec l'expression `instance...where`
- Seuls les types définis avec `data` ou `newtype` peuvent être des instances de classes
- L'implémentation par défaut peut être surchargée

Au minimum, il faut définir les fonctions non implémentées par la classe

```
instance Eq Bool where
  False == False      = True
  True  == True       = True
  _     == _           = False
```

Définition de classe de type (2/2)

- Une classe de type peut être étendue par d'autres classes de type
- La classe « fille » impose une contrainte sur le paramètre de type

La classe Ord a est une extension de la classe Eq a

Le paramètre a doit être une instance de Eq

```
class Eq a => Ord a where
    (<), (<=), (>) , (>=) :: a -> a -> Bool

    min, max :: a -> a -> a
    min x y | x <= y = x
              | otherwise = y
    max x y | x <= y = y
              | otherwise = x
```

```
instance Ord Bool where
    False < True = True
    _ < _ = False
    b <= c = (b < c) || (b == c)
    b > c = c < b
    b >= c = c <= b
```

Instanciation explicite

- Instanciation automatique (ou implicite) des classes Eq, Ord, Show ou Read

```
data Bool = False | True
deriving (Eq, Ord, Show, Read)
```

```
> False == True
False
> False < True
True
> show False
"False"
> read "True" :: Bool
True
```

-- Ordre des constructeurs

-- Inférence de type échoue

Fonctions d'ordre supérieur (1/5)

- Une fonction d'ordre supérieur est une fonction qui prend comme argument une fonction et / ou qui retourne une fonction
- Par définition, une fonction currifiée est une fonction d'ordre supérieur

```
myplus :: (Int -> Int -> Int)
myplus = \x y -> x + y
```

```
twice :: (a -> a) -> a -> a
twice f x = f (f x) -- applique deux fois la fonction f
```

```
> twice reverse [1,2,3]
[1, 2, 3]
> twice (*2) 3
12
> let f = twice reverse
> f "abcdef"
"abcdef"
```

- Composition de fonction

- L'opérateur . retourne la composition de deux fonctions comme une seule fonction

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \x -> f (g x)
```

- L'opérateur de composition simplifie l'application emboîtée de fonction

```
twice f x = f (f x) ⇔ twice f = f . f
```

```
odd n = not (even n) ⇔ odd = not . even
```

```
> twice (*2) 2
8
> twice (map (*2)) [1,2,3]
[4, 8, 12]
```

Fonctions d'ordre supérieur (2/5)

- Les fonctions d'ordres supérieurs sur les listes

-- applique la fonction à chaque élément

-- et retourne la liste des résultats

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

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

> map (+1) [1,2,3]

[2,3,4]

> map even [1,2,3,4]

[False, True, False, True]

-- sélectionne les éléments satisfaisant un

-- prédicat

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

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

> filter (>5) [1..10]

[6,7,8,9,10]

> **all** even [2,4,6,8]

-- Test si tous les éléments satisfont un prédicat

True

> **any** odd [2,4,6,8]

-- Test si au moins un élément satisfait un prédicat

False

> **takeWhile** even [2,4,6,7,8] -- Sélectionne les éléments tant qu'ils satisfont un prédicat

[2, 4, 6]

> **dropWhile** odd [1,3,5,6,7] -- Supprime les éléments tant qu'ils satisfont un prédicat

[6, 7]

Fonctions d'ordre supérieur (3/5)

- Les fonctions de plis de structures de données
 - Beaucoup de fonctions sur les listes sont définies avec une forme récursive identique
 - Le but est de parcourir la structure est de la « plier » en une valeur finale (accumulateur) tenant compte de tous les éléments de la structure

```
-- fonction sum des éléments d'une liste  
sum []      = 0  
sum (x : xs) = x + sum xs
```

```
-- fonction product des éléments d'une liste  
product []     = 1  
product (x : xs) = x * product xs
```

- Cette forme récursive récurrente peut être schématisée de la manière suivante :

```
f []          = v           -- Retourne une valeur v si la liste est vide  
f (x : xs)    = x # f xs   -- Retourne l'application de l'opérateur #  
                           -- L'opérateur # prend la tête et le résultat de l'appel récursif sur le reste
```

- La fonction d'ordre supérieur **foldr** réifie ce schéma de pliage d'une liste

```
foldr :: (a -> b -> b) -> b -> [a] -> b  
foldr f v [] = v           -- Retourne la valeur initiale de l'accumulateur  
foldr f v (x:xs) = f x (foldr f v xs) -- Applique la fonction f sur la tête et le résultat du reste
```

```
> foldr (+) 0 [1,2,3]  
6
```

```
foldr (+) 0 [1,2,3]  ⇔ (+) 1 ( foldr (+) 0 [2,3] )  
                     ⇔ (+) 1 ( (+) 2 (foldr (+) 0 [3] ) )  
                     ⇔ (+) 1 ( (+) 2 ( (+) 3 (foldr (+) 0 []) ) )  
                     ⇔ (+) 1 ( (+) 2 ( (+) 3 0 ) )
```

== 6

Fonctions d'ordre supérieur (4/5)

- Les fonctions de plis de structures de données (suite)

- La fonction *foldr* est un schéma de plis par la droite

- De manière générale, sur une liste de 4 éléments nous avons :

```
foldr f v [x1, x2, x3, x4] == f x1 ( f x2 ( f x3 ( f x4 v ) ) )
```

- Exemples :

```
-- fonction sum des éléments d'une liste
```

```
sum :: Num a => [a] -> a
```

```
sum xs = foldr (+) 0 xs
```

```
-- fonction product des éléments d'une liste
```

```
product :: Num a => [a] -> a
```

```
product xs = foldr (*) 1 xs
```

```
-- fonction length d'une liste
```

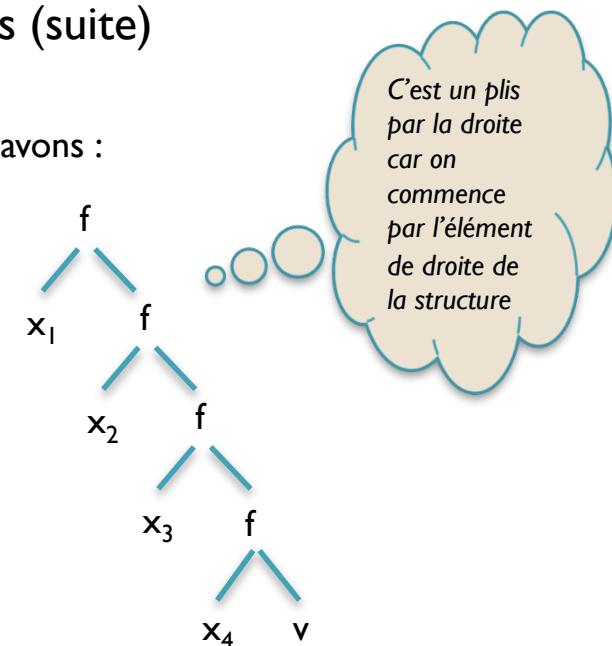
```
length :: [a] -> Int
```

```
length xs = foldr (\_ n -> 1 + n) 0 xs
```

```
-- fonction reverse d'une liste
```

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

```
reverse xs = foldr (\x ys -> ys ++ [x]) [] xs
```



```
[4, 6, 7] ⇔ 4 : (6 : (7 : []))
```

```
length [4, 6, 7] ⇔ 1 + (1 + (1 + 0))
```

```
[1, 2, 3] ⇔ 1 : (2 : (3 : []))
```

```
reverse [1, 2, 3] ⇔ ([] ++ 3) ++ 2 ++ 1
```

Fonctions d'ordre supérieur (5/5)

- Les fonctions de plis de structures de données (suite)
 - Le pliage sur une structure peut aussi être défini pour un plis par la gauche
 - Dans ce cas, nous avons le schéma récursif suivant pour une liste :

```
f v []      = v          -- Retourne une valeur v si la liste est vide
f v (x : xs) = f (v # x) xs -- Retourne l'application récursive de f
                                -- Applique à f le résultat de l'opérateur sur la tête et le reste
```

- La fonction d'ordre supérieur **foldl** réifie le schéma de pliage d'une liste par la gauche :

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f v [] = v          -- Retourne la valeur initiale de l'accumulateur
foldl f v (x:xs) = foldl f (f v x) xs -- Applique à foldl le résultat de f sur la tête et le reste
```

- De manière générale sur une liste de 4 éléments nous avons :

```
foldl f v [x1, x2, x3, x4] == f ( f ( f ( f v x1) x2) x3) x4
```

- Exemples

```
sum xs = foldl (+) 0 xs
```

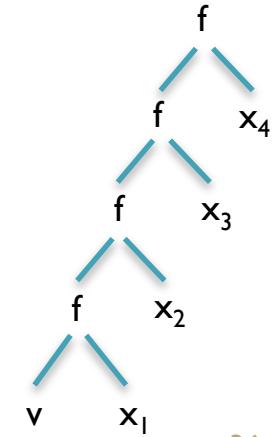
foldl (+) 0 xs == foldr (+) 0 xs
Pourquoi ?

```
length xs = foldl ( \v _ -> v + 1) 0 xs
```

```
length [4,5,6] ⇔ ( (0 + 1) + 1 ) + 1
```

```
reverse xs = foldl ( \v x -> x : v ) [] xs
```

```
reverse [1,2,3] ⇔ 3 : (2 : (1 : []))
```



Les fonctions impures (1/2)

- Les fonctions impures sont des fonctions qui interagissent avec l'extérieur du programme Haskell : elles lisent et / ou écrivent des données
- Une fonction impure est vue comme une fonction prenant en entrée un état du « monde » et retournant en sortie un état modifié du « monde »
- Une action sur le « monde » est définie par le type polymorphe *IO*

```
type IO a = World -> (a, World)    -- Attention : ceci n'est pas la vraie définition du type IO
```

- Par exemples :
 - *IO Char* est une action sur le monde qui retournera une valeur de type *Char*
 - *IO ()* est une action sur le monde mais qui ne retournera rien
- Une action *IO* est exécutée soit par l'interpréteur soit par la fonction *main*
- Une action *IO* peut être exécutée plusieurs fois !
- Une fonction impure = une fonction qui renvoie une valeur de type *IO*
- Fonctions impures basiques : lecture et écriture d'un caractère

```
-- Lire un caractère depuis l'entrée standard, l'afficher et le retourner  
getChar :: IO Char
```

```
-- Ecrire un caractère dans la sortie standard  
putChar :: Char -> IO ()
```

Les fonctions impures (2/2)

- La fonction **return** : passer d'une expression pure à une expression impure

```
return :: a -> IO a           -- Encapsule un type quelconque a dans un type IO
```

- L'expression **do** ≈ permet d'enchaîner simplement des fonctions impures

-- exemple d'utilisation de l'expression do avec le type *IO*

```
act :: IO (Char, Char)
```

```
act = do
```

```
  x <- getchar
```

```
  getchar
```

```
  y <- getchar
```

```
  return (x, y)
```

-- Lit puis donne accès à la valeur retournée via *x* ; *x* est un Char

-- Lit mais ne donne pas accès à la valeur retournée

-- Lit puis donne accès à la valeur retournée via *y* ; *y* est un Char

-- Transforme le tuple (*x*, *y*) en action *IO* (*x*, *y*)

Pour une explication précise, cf. la classe de type Monad ci-après dans le cours

- Quelques exemples de fonctions existantes :

-- Lire une chaîne au clavier

```
getLine :: IO String  
getLine = do x <- getChar  
            if x == '\n'  
            then return []  
            else do xs <- getLine  
                    return (x:xs)
```

-- Ecrire une chaîne

```
putStr :: String -> IO ()  
putStr [] = return ()  
putStr (x:xs) = do  
    putChar x  
    putStr xs
```

-- Ecrire une chaîne avec retour chariot

```
putStrLn :: String -> IO ()  
putStrLn xs = do  
    putStr xs  
    putChar '\n'
```

- Il existe aussi des fonctions pour les fichiers textes ou binaires (cf. doc.)

La classe de type Functor (1/2)

- La classe de type Functor permet de définir des types "mappables"
 - Le type mappable doit être un type polymorphe avec un seul paramètre (\approx un conteneur)
 - Un type mappable est un type sur lequel on pourra appliquer une fonction unaire quelconque
 - Le type mappable est préservé même après application de la fonction unaire
 - Par exemple, le type liste est une instance de Functor

```
class Functor t where
    fmap :: (a -> b) -> t a -> t b
                                -- t : constructeur de type sans son paramètre
                                -- t x : constructeur de type avec son paramètre
```

```
instance Functor [] where
    fmap = map
                                -- [a] est un sucre syntaxiques pour [] a
                                -- la fonction map retourne une liste de résultat
```

- Exemple d'instanciation sur un type arbre binaire

```
data BinaryTree a = EmptyTree | Node a (BinaryTree l) (BinaryTree r)
```

```
instance Functor BinaryTree where
    fmap f EmptyTree = EmptyTree
    fmap f (Node x lxs rxs) = Node (f x) (fmap f lxs) (fmap f rxs)
```



L'implémentation de
fmap préserve le
type BinaryTree

```
> fmap (*2) ( Node 7 ( Node 1 EmptyTree EmptyTree ) ( Node 5 EmptyTree EmptyTree ) )
Node 14 (Node 2 EmptyTree EmptyTree) (Node 10 EmptyTree EmptyTree)
```

La classe de type Functor (2/2)

- Le type Maybe est un Functor

```
instance Functor Maybe where
    fmap _ Nothing = Nothing
    fmap f (Just a) = Just (f a)
```

```
> fmap (+1) Nothing
Nothing
```

```
> fmap (+1) (Just 3)
Just 4
```

La classe Applicative Functor

- Un foncteur applicatif est un foncteur pouvant encapsuler une fonction

```
class Functor t => Applicative t where -- Attention : contrainte sur le type t  
    pure :: a -> t a  
    (<>*) :: t (a -> b) -> t a -> t b -- encapsule un type dans un foncteur applicatif  
                                              -- applique un foncteur applicatif sur un foncteur
```

```
> Just (*3) <*> Just 9  
Just 27  
> pure (+) <*> Just 5 <*> Just 3  
Just 8 -- <*> est associé à gauche
```

- Synonyme de *fmap* en notation infixe : la fonction **<\$>**

```
(<$>) :: Functor t => (a -> b) -> t a -> t b  
f <$> x = fmap f x
```

```
> show <$> Nothing  
Nothing  
> show <$> Just 3  
Just "3"
```

```
> (*2) <$> [1,2,3]  
[2,4,6]
```

```
> (++) <$> Just "John" <*> Just "Travolta"  
Just "John Travolta"
```

- Le type **IO** et le type **[]** sont ces foncteurs applicatifs

```
instance Applicative [] where  
    pure x = [x]  
    fs <*> xs = [ f x | f <- fs, x <- xs ]
```

```
pure (+2) <*> [1,2,3] ⇔ [3,4,5]
```

```
> [(+), (*)] <*> [1,2] <*> [3,4] ⇔ [(+1), (+2), (*1), (*2)] <*> [3,4]  
[4, 5, 5, 6, 3, 4, 6, 8]
```

```
> (*) <$> [2, 5, 10] <*> [8, 10, 11]  
[16, 20, 22, 40, 50, 55, 80, 100, 110]
```

La classe de type Monad (1/2)

- Un type monadique est un foncteur applicatif sur lequel on peut effectuer un traitement en séquence grâce à une fonction de liaison nommée `>>=`

```
class Applicative m => Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
        -- Injecte une valeur quelconque dans une monade
        -- Fonction de liaison servant au traitement en séquence
        -- Transforme une monade en une nouvelle monade
```

- Le type `Maybe` est une instance de `Monad`

La transformation
n'est faite que pour
une monade
« valide »

```
instance Monad Maybe where
    return x = Just x
    Nothing >>= _ = Nothing
    (Just x) >>= f = f x
        -- Transforme une valeur en une valeur de type Maybe
        -- Retourne Nothing
        -- Retourne le résultat monadique de la fonction
```

- Exemple d'enchainement monadique

```
type Birds = Int
type Pole = (Birds, Birds)
landLeft :: Birds -> Pole -> Maybe Pole
landLeft n (left, right)
| abs ((left + n) - right) < 4 = Just (left + n, right)
| otherwise = Nothing
```

```
> return (0,0) >>= landRight 2 -- Just (0,2)
    >>= landLeft 2   -- Just (2,2)
    >>= landRight 2
Just (2,4)
```

```
> return (0,0) >>= landLeft 1   -- Just (1,0)
    >>= landRight 4   -- Just (1,4)
    >>= landLeft (-1) -- Nothing
    >>= landRight (-2)
```

Nothing est propagé dans la séquence

Nothing

La classe de type Monad (2/2)

- Traitement en séquence d'un type monadique : l'expression **do**

L'expression **do** renvoie la dernière expression. Elle est donc du type de cette dernière expression.

```
> return (0,0) >>= landRight 2  
      >>= landLeft 2  
      >>= landRight 2  
Just (2,4)
```

L'expression **do** s'applique à tous types monadiques

```
routine :: Maybe Pole  
routine = do  
  start    <- return (0,0)  
  first    <- landRight 2 start  
  second   <- landLeft 2 first  
  landRight 2 second
```

Le type IO est une instance de Monad !

- Le type liste est une instance de Monad

```
instance Monad [] where  
  -- (>>=) :: [a] -> (a -> [b]) -> [b]  
  xs >>= f = [ y | x <- xs, y <- f x ]
```

```
> [3,4,5] >>= \x -> [x, -x]  
[3, -3, 4, -4, 5, -5]
```

```
-- on utilise une liste par compréhension  
paires :: [a] -> [b] -> [(a,b)]  
paires xs ys = [ (x, y) | x <- xs, y <- ys ]
```

Une liste par compréhension est en fait un succeur syntaxique !

```
> paires [1,2] [3,4]  
[(1,3), (1,4), (2,3), (2,4)]
```

```
paires :: [a] -> [b] -> [(a,b)]  
paires xs ys = do  
  x <- xs  
  y <- ys  
  return (x,y)
```