

M1 MIAGE Option RIM

Cours 2 : Haskell

Alexandre Termier

2010-2011 S2

Haskell

- Langage fonctionnel
- Pur
- Paresseux (Lazy)



Haskell B. Curry
(1900-1982)

- Programmation fonctionnelle
 - Calcul = résultat d'une fonction (au sens mathématique)
 - Pas d'états ni d'objets modifiables
 - Base : le λ -calcul
- Langage fonctionnel pur
 - Impossible de faire des “updates” sur un objet
 - Pas de pointeurs !
- Langage fonctionnel paresseux
 - Evaluation d'un calcul uniquement si nécessaire, et au dernier moment

```
x = ...calcul très long...  
affiche ( "hello world" )
```

```
lst = [0...+infini]  
affiche ( 3 premiers éléments de lst )
```

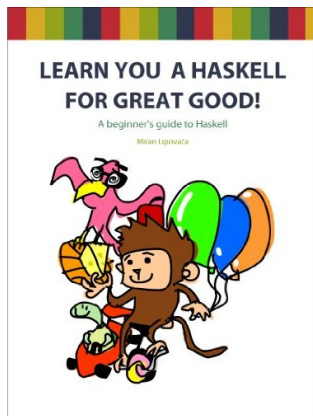
```
lst2 = [2.0, 4/0, 98*1000]  
affiche ( longueur lst2 )
```

Langages fonctionnels dans l'industrie

- Ericsson → langage Erlang
 - T-Mobile
 - Nortel
 - Facebook
- OCaml
 - Applications financières
 - Programmation de robots industriels
- Haskell
 - Aéronautique
 - Programmation web
 - Google : Haskell in industry
- Microsoft → F#

Pourquoi Haskell ?

- Très expressif : programmes courts \rightarrow prototypage
- Typage (très) fort + pur : fiable
- Tests faciles dans l'interpréteur
- Options d'optimisations via le compilateur
- Nombreuses fonctions pré-définies
- Parallélisme



Tutoriel en ligne : <http://learnyouahaskell.com>

Premier contact avec Haskell

- Notre nouvel ami : l'interpréteur Haskell
 - Execution de commandes / programmes
 - Apprendre le langage
 - Tester
 - Débugger
 - Lancé avec : `ghci`

Exemple (ghci)

```
$ ghci-6.12.3
GHCi, version 6.12.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
Prelude>
```

- Prelude : "Package" de base Haskell
 - Calculs simple
 - Opérations sur les listes
 - I/O de base

Example (Calculs)

```
Prelude> 30000 / 125
240.0
Prelude> it * 1.186
284.64
Prelude> let yenEuro = 125.0
Prelude> 9990 / yenEuro
79.92
Prelude> 500/yenEuro >= 75
False
```

- `let variable = expression`

Example (fonctions)

```
Prelude> let convertYenEuro prix = prix / 125
Prelude> convertYenEuro 9990
79.92
Prelude> let applyTax prix = prix * 1.186
Prelude> applyTax it
94.78511999999999
Prelude> applyTax (convertYenEuro 9990)
94.78511999999999
Prelude> applyTax $ convertYenEuro 9990
94.78511999999999
Prelude> (applyTax . convertYenEuro) 9990
94.78511999999999
```

- `let fonction paramètres = expression`
- `$` : application de fonction, `.` : composition de fonctions

- Haskell : langage fortement typé
- *Quand on connaît le type d'une fonction, on peut avoir une idée de ce qu'elle fait !*

Exemple (types de l'ex. précédent)

```
Prelude> :t yenEuro  
yenEuro :: Double
```

- Haskell : langage fortement typé
- *Quand on connaît le type d'une fonction, on peut avoir une idée de ce qu'elle fait !*

Exemple (types de l'ex. précédent)

```
Prelude> :t yenEuro  
yenEuro :: Double
```

```
Prelude> :t convertYenEuro  
convertYenEuro :: (Fractional a) => a -> a  
Prelude> :t applyTax  
applyTax :: (Fractional a) => a -> a
```

- Haskell : langage fortement typé
- *Quand on connaît le type d'une fonction, on peut avoir une idée de ce qu'elle fait !*

Exemple (types de l'ex. précédent)

```
Prelude> :t yenEuro  
yenEuro :: Double
```

```
Prelude> :t convertYenEuro  
convertYenEuro :: (Fractional a) => a -> a  
Prelude> :t applyTax  
applyTax :: (Fractional a) => a -> a
```

```
Prelude> applyTax 15  
17.79
```

- Haskell : langage fortement typé
- *Quand on connaît le type d'une fonction, on peut avoir une idée de ce qu'elle fait !*

Exemple (types de l'ex. précédent)

```
Prelude> :t yenEuro  
yenEuro :: Double
```

```
Prelude> :t convertYenEuro  
convertYenEuro :: (Fractional a) => a -> a  
Prelude> :t applyTax  
applyTax :: (Fractional a) => a -> a
```

```
Prelude> applyTax 15  
17.79
```

```
Prelude> applyTax (15 :: Int)
```

```
<interactive>:1:1:  
  No instance for (Fractional Int)  
    arising from a use of 'applyTax' at <interactive>:1:1-20  
  Possible fix: add an instance declaration for (Fractional Int)  
  In the expression: applyTax (15 :: Int)  
  In the definition of 'it': it = applyTax (15 :: Int)
```

Example (if...then...else...)

```
Prelude> let divDeux n = if (n 'mod' 2 == 0)
                        then n 'div' 2
                        else (n-1) 'div' 2

Prelude> divDeux 200
100
Prelude> divDeux 201
100
```

Example (type de if...then...else...)

```
Prelude> :t (if True then 'a' else 'b')  
(if True then 'a' else 'b') :: Char
```

```
Prelude> let divDeux' n = (if (n `mod` 2 == 0)  
                           then n `div` 2  
                           else (n-1) `div` 2) * 10
```

```
Prelude> divDeux' 200  
1000
```

```
Prelude> divDeux' 201  
1000
```

- `[...]` : définit une liste
`[]` : liste vide

Exemple (liste)

```
Prelude> [1,2,3]  
[1,2,3]
```

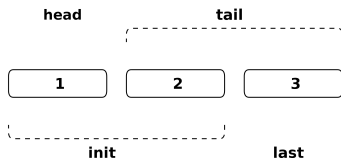
- `:` : ajoute un élément en tête de liste

Exemple (ajout)

```
Prelude> 1:[2,3]  
[1,2,3]  
Prelude> (:) 1 [2,3]  
[1,2,3]
```

- Pour Haskell, `[1,2,3] = 1:2:3:[]`

- Découpage d'une liste



Example (découpages)

```
Prelude> head [1,2,3]
1
Prelude> tail [1,2,3]
[2,3]
Prelude> init [1,2,3]
[1,2]
Prelude> last [1,2,3]
3
```

- ++ : ajoute deux listes

Example (ajout)

```
Prelude> [1,2,3]++[4,5,6]
[1,2,3,4,5,6]
Prelude> "Hello"++" "++"world !"
"Hello world !"

Prelude> :t (++)
(++) :: [a] -> [a] -> [a]
```

- `!!` : accède au *i*ème élément

Exemple (`!!`)

```
Prelude> "Hello" !! 0
```

```
'H'
```

```
Prelude> "Hello" !! 1
```

```
'e'
```

```
Prelude> "Hello" !! 2
```

```
'l'
```

```
Prelude> :t (!!)
```

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

- `length` : longueur d'une liste

Example (`length`)

```
Prelude> length [1,2,3]
```

```
3
```

```
Prelude> length "Hello"
```

```
5
```

```
Prelude> length [[],[1,2,3]]
```

```
2
```

```
Prelude> :t length
```

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

- `take` : renvoie les N premiers éléments d'une liste

Example (take)

```
Prelude> take 2 [1,2,3,4,5]
[1,2]
Prelude> take 0 [1,2,3,4,5]
[]
Prelude> take 100 [1,2,3,4,5]
[1,2,3,4,5]

Prelude> :t take
take :: Int -> [a] -> [a]
```

- `elem` : teste la présence d'un élément dans une liste

Example (`elem`)

```
Prelude> 4 'elem' [1,2,3,4,5]
```

```
True
```

```
Prelude> 12 'elem' [1,2,3,4,5]
```

```
False
```

```
Prelude> :t elem
```

```
elem :: Eq a => a -> [a] -> Bool
```

Domaines (ranges en anglais)

- $[N..M]$ ($N < M$) : renvoie la liste de tous les éléments entre N et M (si c'est possible)

Example (domaines)

```
Prelude> [1..10]
[1,2,3,4,5,6,7,8,9,10]
Prelude> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
Prelude> [10..2]
[]
```

Compréhensions de listes (1/2)

- Construction avancée de listes :
 - on spécifie les données de départ
 - on donne des conditions à vérifier
 - on donne des traitements à faire sur les valeurs obtenues

Exemple (compréhension de liste)

```
Prelude> let lstPrixHT = [100.0,200.0,175.0,345.0]
Prelude> [prix * 1.186 | prix <- lstPrixHT]
[118.6,237.2,207.54999999999998,409.16999999999996]
```

```
Prelude> [prix * 1.186 | prix <- lstPrixHT, prix < 200.0]
[118.6,207.54999999999998]
```


Compréhensions de listes (2/2)

- `[(x,y) | x <- liste1, y <- liste2]`
 - `(x,y)` représente alors le produit cartésien de `liste1` et `liste2`

Exemple (produit cartésien)

```
Prelude> let noms = ["Frodon", "Sam", "Golum"]
Prelude> let adjectifs = ["paresseux", "temeraire", "brave"]
Prelude> [n++" le "++a | n <- noms, a <- adjectifs]
["Frodon le paresseux", "Frodon le temeraire", "Frodon le brave",
"Sam le paresseux", "Sam le temeraire", "Sam le brave",
"Golum le paresseux", "Golum le temeraire", "Golum le brave"]

Prelude> [n++" le "++a | n <- noms, a <- adjectifs,
           (n,a) /= ("Frodon", "paresseux")]
["Frodon le temeraire", "Frodon le brave",
"Sam le paresseux", "Sam le temeraire", "Sam le brave",
"Golum le paresseux", "Golum le temeraire", "Golum le brave"]
```

- Ensemble de valeurs de taille fixe, types peuvent être différents

Example (tuples)

```
Prelude> let star = ("Marylin",1926)
Prelude> :t star
star :: ([Char], Integer)
Prelude> fst star
"Marilyn"
Prelude> snd star
1926
```

- On peut faire des listes de tuples, par ex. [(a,b)]
- zip permet de combiner facilement deux listes en une liste de tuples

Example (zip)

```
Prelude> zip ["Marylin", "Monica", "Laeticia"]  
            [1926,1964,1978]  
[("Marylin",1926),("Monica",1964),("Laeticia",1978)]
```

- `Int` : entiers (efficace)
- `Integer` : entiers, peut représenter très grands nombres, inefficace
- `Float`, `Double` : nombres réels simple et double précision
- `Char` : caractères, entre simple quotes : `'a'`
- `Bool` : booléens \rightarrow `True` et `False`

Classes de types (typeclass)

- Eq : types sur lesquels on peut tester l'égalité

Example (Eq)

```
Prelude> :t (==)
(==) :: Eq a => a -> a -> Bool
Prelude> :t (/=)
(/=) :: Eq a => a -> a -> Bool
```

- Ord : types sur lesquels il y a un ordre

Example (Ord)

```
Prelude> :t (<)
(<) :: Ord a => a -> a -> Bool
Prelude> :t (>)
(>=) :: Ord a => a -> a -> Bool
```

Classes de types (typeclass)

- Show : types qui peuvent être affichés dans une String par show

Example (show)

```
Prelude> :t show
show :: Show a => a -> String
Prelude> show 34.5
"34.5"
```

- Read : inverse de Show, types qu'on peut convertir à partir d'une String avec read

Example (read)

```
Prelude> :t read
read :: Read a => String -> a
Prelude> read "24" * 2
48
```

Classes de types (typeclass)

- Num : types qui se comportent comme des nombres.

Exemple (Num)

```
Prelude> :t (+)
```

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

- Integral : types représentant les entiers : Int et Integer
- Floating : types représentant les réels : Float et Double

Exemple (fromIntegral)

```
Prelude> :t length
```

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

```
Prelude> (length [1,2,3]) + 2.5
```

```
--- ERREUR DE TYPE ---
```

```
Prelude> :t fromIntegral
```

```
fromIntegral :: (Integral a, Num b) => a -> b
```

```
Prelude> fromIntegral (length [1,2,3]) + 2.5
```

```
5.5
```

- `map` : appliquer une fonction à tous les éléments d'une liste

Example (`map`)

```
Prelude> :t map
map :: (a -> b) -> [a] -> [b]
Prelude> map (\x -> x*2) [1,2,3]
[2,4,6]
Prelude> map (*2) [1,2,3]
[2,4,6]
Prelude> map (==3) [1,2,3]
[False,False,True]
```


- `foldl` : faire une opération qui accumule tous les résultats d'une liste

Example (foldl)

```
Prelude> :t foldl
foldl :: (a -> b -> a) -> a -> [b] -> a
Prelude> foldl (+) 0 [1,2,3]
6
Prelude> foldl (\acc x -> acc * x) 1 [1,2,3,4]
24
```

Ouverture du fichier

```
Prelude> fichier <- readFile "../Data/LesMiserables_T1.txt"
Prelude> :t fichier
fichier :: String
```

Retour aux Misérables

Ouverture du fichier

```
Prelude> fichier <- readFile "../Data/LesMiserables_T1.txt"  
Prelude> :t fichier  
fichier :: String
```

Découpage en lignes

```
Prelude> :t lines  
lines :: String -> [String]
```

Retour aux Misérables

Ouverture du fichier

```
Prelude> fichier <- readFile "../Data/LesMiserables_T1.txt"
Prelude> :t fichier
fichier :: String
```

Découpage en lignes

```
Prelude> :t lines
lines :: String -> [String]

Prelude> let lignesMT1 = lines fichier
Prelude> take 3 lignesMT1
["The Project Gutenberg EBook of Les mis\233rables Tome I, by Victor Hugo","",
"This eBook is for the use of anyone anywhere at no cost and with"]
```

Retour aux Misérables

Ouverture du fichier

```
Prelude> fichier <- readFile "../Data/LesMiserables_T1.txt"
Prelude> :t fichier
fichier :: String
```

Découpage en lignes

```
Prelude> :t lines
lines :: String -> [String]

Prelude> let lignesMT1 = lines fichier
Prelude> take 3 lignesMT1
["The Project Gutenberg EBook of Les mis\233rables Tome I, by Victor Hugo","",
"This eBook is for the use of anyone anywhere at no cost and with"]
```

Découpage en mots

```
Prelude>:t words
words :: String -> [String]

Prelude> let motsMT1 = words fichier
Prelude> take 10 motsMT1
["The","Project","Gutenberg","EBook","of","Les","mis\233rables","Tome","I","","by",
"Victor","Hugo","This","eBook","is","for","the","use","of","anyone"]
```

Example (motSuiv)

```
-- fonction motSuiv : permet de savoir les mot qui suit un autre,  
-- malgré les sauts de ligne  
motSuiv :: String -> [String] -> [String]  
motSuiv m [] = []  
motSuiv m [_] = []  
motSuiv m (x:xs) = if (x == m)  
                    then ((head xs):(motSuiv m xs))  
                    else (motSuiv m xs)
```

Example (motSuiv ghci)

```
:{  
  let { motSuiv m [] = []  
        ; motSuiv m [_] = []  
        ; motSuiv m (x:xs) = if (x == m)  
                                then ((head xs):(motSuiv m xs))  
                                else (motSuiv m xs)  
      }  
:}
```

Example (noms propres)

```
Prelude> :m +Text.Regex.Posix
Prelude Text.Regex.Posix> let getNomsPropres mots =
                        (map (=~ "[A-Z][a-z]*") mots)::[String]
Prelude Text.Regex.Posix> getNomsPropres $ motSuiV "Jean" motsMT1
Loading package array-0.3.0.1 ... linking ... done.
Loading package bytestring-0.9.1.7 ... linking ... done.
Loading package containers-0.3.0.0 ... linking ... done.
Loading package mtl-1.1.0.2 ... linking ... done.
Loading package regex-base-0.93.1 ... linking ... done.
Loading package regex-posix-0.94.1 ... linking ... done.

["Valjean","", "Soanen","", "", "Valjean", "Valjean", .....
```


Example (unicité)

```
Prelude Text.Regex.Posix> let getDistincts mots =  
    Data.Set.toList $  
    foldl (\acc x -> Data.Set.insert x acc)  
    Data.Set.empty mots  
Prelude Text.Regex.Posix> getDistincts $ getNomsPropres  
    $ motSuiv "Jean" motsMT1  
["", "Mathieu", "Soanen", "Valjean"]  
Prelude Text.Regex.Posix> getDistincts $ motSuiv "Jean" motsMT1  
["Mathieu", "Mathieu!", "Mathieu;", "Mathieu?", "Soanen,", "Valjean",  
"Valjean!", "Valjean!--Jean", "Valjean,", "Valjean--nous", "Valjean.",  
"Valjean:", "Valjean;", "Valjean?", "de", "et", "n'eut", "peut", "vous"]
```

Mieux que grep !

Exemple (grep)

```
$ grep -o -E 'devenu [A-Za-z]+' LesMiserables_T1.txt
devenu une
devenu millionnaire
devenu bandit
devenu riche
devenu un
devenu monsieur
devenu presque
devenu riche
devenu maire
devenu tison
devenu fou
```

Exemple (notre programme)

```
*Main> getDistincts $ motSuiv "devenu" motsMT1
["bandit,","capable,","fou,","maire;","millionnaire","monsieur",
"m\233chant;","presque","riche,","tison. ","un","une"]
```