M1 MIAGE Option IFD
Cours 3 : Haskell (fonctions)

Alexandre Termier

2011-2012 S2

Fonctions : la base

Ecrire une fonction en Haskell

nomFonction :: typage
nomFonction arguments = expression

Example (Fonction simple)

```
applyTax :: (Fractional a) => a -> a
applyTax prix = prix * 1.186
```

Fonctions : patterns

Pattern matching dans les fonctions

nomFonction :: typage
nomFonction pattern1 = expression1
nomFonction pattern2 = expression2
...

Example (Pattern matching)

```
trouveValjean :: String -> Bool
trouveValjean "Valjean" = True
trouveValjean x = False

Prelude> trouveValjean "Valjean"
True
Prelude> trouveValjean "Cosette"
False
```

Fonctions : patterns

Example (factorielle)

```
factorielle :: (Integral a) => a -> a
factorielle 0 = 1
factorielle n = n * factorielle (n-1)

Prelude> factorielle 0
1
Prelude> factorielle 5
120
Prelude> factorielle 40
815915283247897734345611269596115894272000000000
```

Patterns

Contenu des patterns

- Constantes → `5, "Valjean", []`
- → une valeur qui ne sera pas utilisée
- Variables → utilisées dans l'expression de droite
  - variables peuvent être dans structures :
  - `(x,y)` : tuple
  - `(x:xs)` : liste, `head = x, tail = xs`

Patterns

Example (valeurs non utilisées)

```
trouveValjean2 :: String -> Bool
trouveValjean2 "Valjean" = True
trouveValjean2 _ = False

Prelude> trouveValjean2 "Valjean"
True
Prelude> trouveValjean2 "Cosette"
False
```

Récursion

- Principes
  - Définir un cas de base (1ers patterns)
  - Définir une règle générale à suivre (derniers patterns)
  - La boucle du fonctionnel !

Example (longueur d'une liste)

```
longueur :: (Num b) => [a] -> b
longueur [] = 0
longueur (_:xs) = 1 + longueur xs

Prelude> longueur ["Valjean","Cosette","Myriel"]
3
```

Mots suivants (1/2)

Example (motSuiv)

```
-- fonction motSuiv : permet de savoir les mot
-- en suivant 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)
```

Mots suivants (2/2)

Example (utilisation de motSuiv)

```
*Main> fichier <- readFile "../Data/LesMiserables_T1.txt"
*Main> let motsMT1 = words fichier
*Main> take 10 $ motSuiv "Valjean" motsMT1
["Chapitre","du","Vers","se","était","était","qu'une",
"venait","fut","fut"]
*Main> take 5 $ motSuiv "Jean" motsMT1
["Valjean","peut","Soanen,","vous","de"]
```

Problème dans les mots suivants

grep -A1 -e "Valjean" LesMiserables_T1.txt

Chapitre VI Jean Valjean
Chapitre VII Le dedans du désespoir
–
nom? Vous vous appelez Jean Valjean. Maintenant voulez-vous que je vous dise qui vous êtes? En vous voyant entrer, je me suis douté de quelque
–
–Voici. Je m'appelle Jean Valjean. Je suis un galérien. J'ai passé dix-neuf ans au bagne. Je suis libéré depuis quatre jours et en route

- Erreur dans motSuiv ?

Expressions rationnelles en Haskell

Package Text.Regex.Posix

- Invocation
  - Dans ghci : `:m +Text.Regex.Posix`
  - Dans un fichier .hs : `import Text.Regex.Posix`
- Utilisation
  - chaîne =~ pattern + type du résultat attendu

Utilisation des expressions rationnelles

- `Bool` : y'a t'il un match ou non ?

Example (Bool)

```
> "Valjean" =~ "(Valjean|Cosette)" :: Bool
True
> "Cosette" =~ "(Valjean|Cosette)" :: Bool
True
> "Jean-Pierre" =~ "(Valjean|Cosette)" :: Bool
False
```

Utilisation des expressions rationnelles

- `String` : première chaîne qui matche

Example (String)

```
> "--Voici. Je m'appelle..." =~ "[A-Za-z]+" :: String
"Voici"

ATTENTION !

>  "--Voici. Je m'appelle..." =~ "[A-Za-z]*" :: String
""
```

Utilisation des expressions rationnelles

- `[ [ String ] ]` : toutes les chaînes qui matchent

Example (toutes les chaînes)

```
> "--Voici. Je m'appelle..." =~ "[A-Za-z]+" :: [[String]]
[["Voici"],["Je"],["m"],["appelle"]]

> "les poules du couvent couvent" =~ "couvent" ::[[String]]
[["couvent"],["couvent"]]
```

Utilisation des expressions rationnelles

- `Int` : nombre de matchs

Example (Int)

```
> "--Voici. Je m'appelle..." =~ "[A-Za-z]+" :: Int
4

> "aaaa" =~ "aa" :: Int
2
```

Retour aux mots suivants

Eliminer la ponctuation ?

Retour aux mots suivants

Eliminer la ponctuation ?

```
let virePonctuation mots =
    (map (=~ "[A-Za-z]+") mots)::[String]
```

Retour aux mots suivants

Eliminer la ponctuation ?

```
let virePonctuation mots =
    (map (=~ "[A-Za-z]+") mots)::[String]
```

Example (Résultat)

```
> take 10 $ motSuiv "Valjean" $ virePonctuation motsMT1
["Chapitre","Maintenant","Je","for","c","une","n","du",
"Vers","se"]
```

Gardes

Utilité

- Permettent de tester rapidement des conditions sur les paramètres
- Equivalent à une cascade de if...else..., en plus lisible

Example (gardes)

```
pulsations :: Int -> String
pulsations p
  | p <= 40 = "Choquez le !"
  | p <= 70 = "Vous dormez ?"
  | p <= 120 = "Ca va."
  | p <= 180 = "Vous avez couru ?"
  | otherwise = "Un valium, vite !"
```

Gardes

Example (test gardes)

```
> pulsations 0
"Choquez le !"
> pulsations 200
"Un valium, vite !"
> pulsations 123
"Vous avez couru ?"
> pulsations 100
"Ca va."
```

Variables et fonctions locales

Deux méthodes

- Définition en fin de fonction : `where`
- Définition en début de fonction : `let`

Example (where)

```
calculTaxes prix = applyTax prix
  where applyTax p = tx * p
        tx = 1.186

> calculTaxes 150
177.89999999999998
```

Variables et fonctions locales

Example (let)

```
calculTaxes2 prix =
  let tx = 1.186
      applyTax p = tx*p
  in  applyTax prix

> calculTaxes2 150
177.89999999999998
```

Différence let/where

- `where...` = sucre syntaxique
- `let...in...` est une expression a part entière

Example (expression avec let)

```
> 3*(let applyTax p = 1.186*p in applyTax 150)
533.6999999999999
```