

Les bases de Haskell

Programmation Fonctionnelle
Master 2 I2L apprentissage

SÉBASTIEN VEREL

verel@univ-littoral.fr

<http://www-lisic.univ-littoral.fr/~verel>

Université du Littoral Côte d'Opale
Laboratoire LISIC
Equipe OSMOSE

Septembre 2019

Plan

- 1 Calculatrice
- 2 Types composés
- 3 Pattern matching et fonction
- 4 Structures conditionnelles

Avertissements

Ce premier cours n'est pas un cours à proprement parler un cours de programmation fonctionnelle

Introduction à la syntaxe d'un langage fonctionnel,
le langage Haskell

Pourquoi Haskell ?

- Langage paradigme de **programmation fonctionnelle** "pur"
- Possibilité de faire, en autres, des frameworks web, cf. cours semestre 2
- Evaluation paresseuse
- Typage statique, inférence de type

Historique

Historique

- ???

Bibliographe / Webographie

- Cours de Julien Dehos (LISIC, M1 isidis) :
<http://www-lisic.univ-littoral.fr/~dehos/>
- <http://haskell.fr/lyah/>
"Ce travail est une traduction de Learn You a Haskell For Great Good!. Le texte original est de Miran Lipovaca, la traduction a été réalisée par Valentin Robert. Le texte original est distribué sous licence Creative Commons Paternité - Pas d'Utilisation Commerciale - Partage à l'Identique 3.0 non transcrit parce que son auteur n'a pas trouvé de licence avec un nom encore plus long. Ce travail est par conséquent redistribué sous la même licence."

Start / Stop

Lancement

Lancer le shell Haskell par :

```
ghci
```

Expression

Une expression est terminée par un retour chariot

```
:quit
```

Commandes

Les commandes Haskell commencent par les deux points :

Trouver d'autres commandes disponibles.

Nombres

Entiers

Classique :

801

-10

Nombres

Flottants

17.5

-5.0

6.022E23

3.02E-2

Précision sur 64 bits ???

Opérateurs arithmétiques

Opérateur	description	types
+	addition	Entier, flottants
-	soustraction	Entier, flottants
*	multiplication	Entier, flottants
/	division à virgule flottante	Entier, flottants
div	division entière	Entiers
mod	reste de la div entière	Entiers
max	maximum de 2 nombres	Entier, flottants
min	minimum de 2 nombres	Entier, flottants
succ	nombre + 1	Entier, flottants

Opérateurs arithmétiques : tests

Exemples

(en notation préfixée)

```
div 13 4
```

```
mod 13 4
```

(en notation infixée)

```
3 'div' 4
```

```
13 'mod' 4
```

A tester (qu'est qui se passe?)

```
succ succ 10
```

A vous !

Exercice

Calculer (le rapidement possible) les nombres suivants :

A vous !

Exercice

Calculer (le rapidement possible) les nombres suivants :

- a $1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10$
- b le carré de l'approximation à 10^{-5} du nombre π
- c Le nombre de secondes de l'année 2017
- d Le nombre d'atomes contenus dans une mole d'eau
- e Le temps (à la minute près) mis par un véhicule pour aller jusqu'à la lune en ligne droite depuis la Terre à la vitesse moyenne de 260 km/h.
- f $22/7$

Booléens

Syntaxe

True et False avec les opérateurs booléens

Opérateur	description
not	négation
&&	et logique
	ou logique

Types

Quelques types principaux

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

Connaitre le type d'un élément avec la commande :
`:type True`

- Typage statique : déterminer à la compilation
- Inférence de type : le type est calculé
il n'est pas nécessaire de le préciser !
- Existence de classe de types (ensemble de types, interface) supportant certaines opérations

Types

```
:t 32
```

```
32 :: Num p => p
```

"32 a pour type p qui doit être dans la classe Num"

```
:t (==)
```

```
(==) :: Eq a => a -> a -> Bool
```

Avant =>, contrainte de classe

"le type a doit être de la classe Eq"

Ecrire un code

- On peut écrire du code en mode interactif avec `ghci`
- Ecrire un code dans un fichier (en général extension `.hs`)
 - Compiler/exécuter depuis le mode interactif avec la commande :
`:load nomDuProgramme`
 - Compiler/exécuter depuis une console :
`runghc nomDuProgramme`
- Fonction `main`, et le `do` permet d'enchaîner plusieurs instructions

A tester avec un fichier `hello.hs`

```
main = do
  putStrLn "- Hello world!"
  putStrLn "- Pardon ?"
  putStrLn "- Bonjour Monsieur."
  putStrLn "- Bonjour Valentino."
```

Les commentaires !

```
-- pour une ligne
```

```
{- ... -} pour un block (imbriquable)
```

Variables

Syntaxe

- Commence par une minuscule
- ensuite lettres, chiffres, caractères soulignés, prime

"Affection" unique

Dans la portée d'une fonction (et donc processus du shell)
On ne peut pas modifier une variable en Haskell
(bizarre non pour une variable?)

Vocabulaire

`a=3`

On ne dit plus ni affectation, ni assignement, mais définition (sous entendue "éternelle").

"a est définie à 3".

Exemple variables

Test immédiat dans un programme

```
a = 1
```

```
a = 2
```

```
newA = a + 1
```

```
main = do
```

```
    print a
```

```
    print newA
```

```
    print $ a + 20
```

```
    -- $ permet d'évaluer tout ce qui est à droite
```

Variables

Quelques remarques et précisions

- Tous les appels de variables se font par *valeur*
- Il n'existe pas d'appel par *référence*
- Les variables sont locales à la fonction dans laquelle elles sont liées (notion d'environnement)
- Pas de variables globales

Comparaison de termes

Opérateurs

Opérateur	description
==	égale à
/=	différent de
<=	inférieur ou égale à
<	strictement inférieur à
>=	supérieur ou égale à
>	strictement supérieur à

Types

Les éléments doivent être de même type.

Test immédiat

1 == 1.0

1 /= 1.0

Exercice

Exercice

Pour $x = 5$, $a = (x \geq 12)$, $b = (x \leq 2)$, $c = (x < 6)$.

- Quelle est la valeur des expressions suivantes :
 - $(a \text{ et } b)$ ou c
 - a et $(b \text{ ou } c)$
 - a ou exclusif b
- Quelle est la valeur des expressions pour $x = 13$?

Tuples

Définition

- Type de données composé
- Stocker une collection d'éléments
- Non nécessairement de même type

Syntaxe

- Éléments placés entre parenthèses
- Éléments séparés par des virgules

Tests immédiats

```
(123, "bois")  
("un", (22, "police"), 42)  
()  
(False, "James", "Bond")  
(True, "Louis", "Defunes")
```

Remarque : le type d'un tuple est le produit des types des éléments.

Fonctions pré-définies sur les couples

- `fst` : premier élément
- `snd` : second élément

Tests immédiats

```
fst (9, 13)
```

```
snd (9, 13)
```

Pour le reste,...

Listes

Ce n'est qu'un début...

Définition

- Type de données composé
- Stocker une collection d'éléments
- Nécessairement de **même type**
- Les traitements ne sont pas les mêmes que sur les tuples (voir la prochaine séance !)

Syntaxe

- Éléments placés entre crochets
- Éléments séparés par des virgules

Tests immédiats

```
[ "lundi", "mardi", "mercredi", "jeudi" ]
```

```
[ 456, "cerise" ]
```

```
[]
```

```
[ [ 'a' ], [ [ 'b', [ 'c', 'd', 'e' ], [ 'f' ] ], [ [ 'g' ] ] ]
```

Combien d'éléments dans cette dernière liste ?

Listes : Fonction pré-définies

- n-ième éléments d'une liste :
[0, 10, 20, 30] !! 3
30

Listes de compréhension

```
[1..10]
```

```
[2*x | x <- [1..10] ]
```

```
[2*x | x <- [1..10], mod x 3 == 0 ]
```

```
[x*y | x <- [1..10], y <- [1..10] ]
```

Chaîne de caractères

Caractère

- de type Char
- caractère entre quotes 'e'

Chaîne de caractères

- Pas de type spécifique aux chaînes de caractères
- chaîne de caractères : liste de caractères

Concaténation

++

Tests immédiats

'A'

'A' +32.

'ee'.

['c', 'o', 'c', 'o', 's', 'a', 'n', 's', 'a']

['c', 'o', 'c', 'o'] ++ " " ++ "sans" ++ [' ', 'a']

Entrée / sortie

Remarque

Dans un monde sans effet de bord, les entrées/sorties ne sont pas de la tarte.

Entrée / sortie

Remarque

Dans un monde sans effet de bord, les entrées/sorties ne sont pas de la tarte.

Entrées

`getLine` : lit sur l'entrée standard

Tests immédiats

```
x <- getLine
x
```

Entrées

Quel est le type de `getLine` ?

Entrée / sortie

Remarque

Dans un monde sans effet de bord, les entrées/sorties ne sont pas de la tarte.

Entrées

`getLine` : lit sur l'entrée standard

Tests immédiats

```
x <- getLine
x
```

Entrées

Quel est le type de `getLine` ?
le constructeur `<-` lie le monde extérieur IO avec la définition

Subtile...

```
x = "Le monde de " ++ getLine
```


Entrée / sortie

Sortie

- `putStr` : écrit une chaîne de caractère sur la sortie standard.
- `putStrLn` : écrit une chaîne de caractère et un retour à la ligne sur la sortie standard.

Sortie le retour

- `print` : Écrit sur la sortie standard n'importe quel type instance de `Show`.
Revient à exécuter `show` puis `putStrLn`

Pattern matching

Définition

- Affecter des valeurs à des variables
- Contrôler le flux d'exécution
- Extraire des valeurs à des données de type composé

Syntaxe

Motif = Expression

- **Motif** : expression composé de variables libres ou liées, ainsi que des valeurs littérales (atomes, entiers ou chaînes)
- **Expression** : expression composé de structures de données, de variables liées, d'opérateur arithmétiques et d'appel de fonctions. Mais pas de variables libres.

Pattern matching

Déroulement

Exécute l'expression à droite de `=` puis compare la valeur au motif

- expression et motif doivent avoir la même forme
 - les littéraux doivent être égaux aux valeurs de l'expression
 - si le matching réussit, les variables libres sont liées aux valeurs correspondantes
 - les variables liées doivent avoir les mêmes valeurs que dans l'expression
-
- S'il a réussi, les variables libres sont liées et le résultat est celui de l'expression
 - Si les types ne sont pas compatibles, une erreur est déclenchées
 - S'il a échoué, aucune variable libre n'est liée, une exception est déclenchée

Pattern matching

Test immédiat

$$(30, x, y) = (30, 3, 2)$$
$$(30, x, y) = (30, 3, 2, 21)$$
$$(30, x, y) = (3, 3, 2)$$
$$(30, x, x) = (30, 2, 2)$$
$$[30, x, y] = [30, 2, 2]$$

_ est une variable "bidon" qui marque seulement l'emplacement

Test immédiat

$$(-, -, y) = (30, 3, 2)$$

Fonction

Fonction

- L'entête est composée d'un nom, suivi de paramètres **sans** parenthèses, puis de =

Test immédiat

```
squareArea cote = cote * cote
```

```
triangleArea base height = (base * height) / 2
```

Bonne pratique : en indiquant les types (et classes de types)

```
squareArea :: (Integral a) => a -> a
```

```
squareArea cote = cote * cote
```

```
triangleArea :: (Fractional a) => a -> a -> a
```

```
triangleArea base height = (base * height) / 2
```

Fonction et filtrage (pattern matching)

Fonction

- On peut utiliser le pattern matching pour définir les fonctions
- On définit les clauses possibles

Test immédiat

```
sayMe :: (Integral a) => a -> String
```

```
sayMe 1 = "One!"
```

```
sayMe 2 = "Two!"
```

```
sayMe 3 = "Three!"
```

```
sayMe 4 = "Four!"
```

```
sayMe x = "Not between 1 and 4"
```

```
first :: (a, b, c) -> a
```

```
first (x, _, _) = x
```

```
third :: (a, b, c) -> c
```

```
third (_, _, z) = z
```

Fonction

Exercice

- Ecrire un module `bool` qui contient des fonctions qui calculent les fonctions logiques *non*, *et*, *ou*, *nand*.

Bien sûr, vous n'êtes pas autorisé à utiliser les fonctions pré-définies.

Gardes !

- Evaluation tour à tour jusqu'à trouver une valeur True
- le résultat est alors la dernière expression évaluée

```
myfun x
| condition1 = expression1
| condition2 = expression2
| condition3 = expression3
| otherwise = expressionfinale
```

Test immédiat

```
juryTell :: (RealFloat a) => a -> String
juryTell moyenne
| moyenne < 10 = "Seconde session"
| moyenne <= 12 = "passable"
| moyenne <= 14 = "pas mal"
| otherwise    = "intéressant"
```


Définition locale avec where

- On peut définir localement avec le mot clé `where` à la suite des gardes.
- Les définitions sont visibles par toutes les gardes

Test immédiat

```
juryTell :: (RealFloat a) => a -> a -> a -> String
juryTell n1 n2 n3
  | moyenne < s1 = "Seconde session"
  | moyenne <= s2 = "passable"
  | moyenne <= s3 = "pas mal"
  | otherwise   = "intéressant"
where moyenne = (n1+n2+n3) / 3
      (s1, s2, s3) = (10, 12, 14)
```

Définition locale avec let

- On peut définir localement avec une construction de la forme :
let <liaison> in <expression>
- Les définitions sont visibles que dans expression
- La valeur de la construction est la valeur de <expression>

Test immédiat

```
cylinder :: (RealFloat a) => a -> a -> a
cylinder r h =
    let sideArea = 2 * pi * r * h
        topArea = pi * r ^2
    in sideArea + 2 * topArea
```

Structure if

- Classique pour tout langage, mais ici le else est obligatoire
- Selon la valeur de la garde, la valeur de la structure if est donnée par le then ou le else

```
if Garde
  then <expressionTrue>
  else <expressionFalse>
```

Test immédiat

```
myabs x = if x > 0
          then x
          else -x
```

Case

- Exactement synonyme du filtrage

```
case expression of pattern -> result
                  pattern -> result
                  pattern -> result
```

Test immédiat

```
head' :: [a] -> a
head' [] = error "No head for empty lists!"
head' (x:_) = x

head' :: [a] -> a
head' xs = case xs of [] -> error "No head for empty lists!"
                   (x:_) -> x
```

Structure conditionnelle

Exercice

- Ecrire de 3 manières une fonction qui calcule la parité d'un nombre entier

Exercice

Jeu didactique

Programmer le jeu qui consiste à deviner un nombre entre 1 et 100.