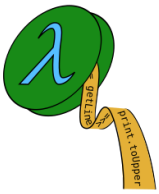

Tuto fonctionnel,

by Nokomprendo

Repo - Stats

RSS - Atom

Introduction au langage Haskell

Voir aussi : vidéo peertube - vidéo youtube - dépôt git

Généralités sur haskell

- programmation fonctionnelle : imbrication de fonctions sans effet de bord
- intérêts : sûreté, expressivité, optimisation du compilateur
- utilisation : compilation, DSL, web backend…
- particularités d'Haskell : fonctionnel pur, évaluation paresseuse
- quelques liens :
  - site officiel
  - wikibook
  - livre Apprendre Haskell vous fera le plus grand bien !
  - livre Real World Haskell
  - What I Wish I Knew When Learning Haskell
  - State of the Haskell ecosystem
  - Typeclassopedia

Notions de bases

Quelques définitions

- valeur : donnée de base

```
Prelude> 42
42
Prelude> 13.37
13.37
Prelude> "toto"
"toto"
```

- expression : morceau de code pouvant être évalué

```
Prelude> 21*2
42
Prelude> "hello" ++ "world"
"helloworld"
```

- type : ensemble prédéfini de valeurs possibles avec des caractéristiques et opérations particulières

```
Prelude> 21*2 :: Int
42
Prelude> "toto" :: String
"toto"
```

- variable : nom auquel on associe une donnée

```
Prelude> x = 12
Prelude> y = "toto" :: String
Prelude> x*2
24
Prelude> y ++ y
"totototo"
```

Quelques remarques

- En Haskell, les variables sont immuables et les expressions respectent la propriété de transparence référentielle
- Haskell calcule les types automatiquement mais on peut l'indiquer (pour plus de lisibilité)
- Haskell est sensible à l'indentation
- les variables et les fonctions commencent par une lettre minuscule
- les types et les classes de type commencent par une lettre majuscule
- les parenthèses servent à définir des priorités d'évaluation
- opérateur d'évaluation `$`
- commentaire de fin de ligne :

```
x = 42  -- ceci est un commentaire
```

- commentaire multi-ligne :

```
{-
ceci est
un commentaire
-}
```

Types de base

- types simples : `Int`, `Float`, `String`...

```
x :: Int
x = 42

y :: String
y = "toto"
```

- types composés : tuples, listes

```
x :: (Int, String)
x = (42, "toto")

y :: [Int]
y = [42, 13, 37]
```

Notion de fonction

- fonction : expression contenant des paramètres; définition/évaluation

```haskell
-- définit une fonction "ajouter"
ajouter :: Int -> Int -> Int
ajouter x y = x + y

-- évalue la fonction "ajouter" pour les paramètres 10 et 32
x :: Int
x = ajouter 10 32
```

- fonction anonyme (lambda)

```haskell
-- définit une lambda et l'évalue pour le paramètre 32
x :: Int
x = (\ n -> 10+n) 32
```

Définir des fonctions par décomposition

- `if-then-else` : définit une expression en décomposant 2 cas

```haskell
formaterParite :: Int -> String
formaterParite x = if even x then "pair" else "impair"
```

- `case` : définit une expression en décomposant plusieurs cas

```haskell
formaterNombre :: Int -> String
formaterNombre x = case x of
  0 -> "zero"
  1 -> "un"
  otherwise -> "plusieurs"
```

- pattern matching : définit une fonction selon la valeur des paramètres

```haskell
formaterNombre :: Int -> String
formaterNombre 0 = "zero"
formaterNombre 1 = "un"
formaterNombre _ = "plusieurs"
```

- gardes : définit une fonction en testant ses paramètres

```haskell
formaterNombre' :: Int -> String
formaterNombre' x
  | x == 0    = "zero"
  | x < 0     = "negatif"
  | otherwise = "positif"
```

Récursivité

- fonction récursive : fonction définie d'après elle-même

```haskell
factorielle :: Int -> Int
factorielle 1 = 1                          -- cas terminal
factorielle x = x * factorielle (x - 1)  -- appel récursif
```

- fonction récursive terminale : l'appel récursif correspond complètement à la dernière expression

```haskell
factorielle :: Int -> Int
factorielle x = aux 1 x
  where aux acc 1 = acc  -- fonction auxiliaire récursive terminale
        aux acc n = aux (acc*n) (n-1)
```

Fonction d'ordre supérieur

- définition : fonction qui retourne une fonction

```haskell
ajouter :: Int -> Int -> Int
ajouter x = \ y -> x + y

-- autre façon de définir :
-- ajouter x y = x + y
```

- évaluation partielle : évaluation d'une fonction pour une partie des paramètres; produit une fonction

```haskell
ajouter42 :: Int -> Int
ajouter42 = ajouter 42  -- ici seul le paramètre "x" de la fonction "ajouter" est donné

-- autre façon de définir ajouter42 :
ajouter42 y = ajouter 42 y
```

- composition de fonctions : définit une fonction en combinant deux autres fonctions

```haskell
fois2 x = x*2
plus1 x = x+1
fois2plus1 = plus1 . fois2

-- autre façon de définir fois2plus1 :
-- fois2plus1 x = plus1 (fois2 x)
```

- notation "point-free" : définit une fonction en combinant d'autres fonctions plutôt qu'en exprimant le calcul sur un point du domaine de définition

```haskell
fois2plus1 x = plus1 (fois2 x)  -- calcul sur un point "x" du domaine
fois2plus1 = plus1 . fois2      -- notation point-free

ajouter42 y = ajouter 42 y  -- calcul sur un point "y"
ajouter42 = ajouter 42      -- notation point-free
```

Fonctions sur des listes

- opérateur de construction de liste, par exemple `(:) :: Int -> [Int] -> [Int]`

```haskell
Prelude> 42:[13,37]
[42,13,37]
```

- pattern matching + récursivité

```
doubler :: [Int] -> [Int]
doubler [] = []
doubler (x:xs) = (2*x):(doubler xs)
```

- mapping : applique une fonction sur chaque élément d'une liste

```
doubler :: [Int] -> [Int]
doubler = map (\ x -> x*2)

-- encore plus concis :
-- doubler = map (*2)
```

- filtrage : conserve les éléments d'une liste qui vérifient un prédicat

```
pairs :: [Int] -> [Int]
pairs = filter (\ x -> even x)

-- encore plus concis :
-- pairs = filter even
```

- réduction : réduit une liste en une valeur en appliquant successivement une fonction sur les éléments

```
somme :: [Int] -> Int
somme = foldl (\ acc x -> acc + x) 0

-- encore plus concis :
-- somme = foldl (+) 0
```

Listes en compréhension

- définition de liste combinant génération + filtrage + mapping

```
Prelude> [(x,y) | x<-[1..10], y<-[x..10], x+y==12]
[(2,10),(3,9),(4,8),(5,7),(6,6)]
```

Modules

- exemple de définition (fichier `Doubler.hs`) :

```
module Doubler where
doubler x = x * 2
```

- exemple d'utilisation :

```
import Doubler
x = doubler 21
```

Types et classes

Type algébriques

- définir un nouveau type :

```
data Forme = Carre Float
           | Rectangle Float Float
```

- créer des valeurs du nouveau type :

```
monCarre = Carre 2.0
monRectangle = Rectangle 2.0 0.5
```

- écrire une fonction avec pattern matching sur les valeurs du type :

```
calculerSurface :: Forme -> Float
calculerSurface (Carre cote) = cote*cote
calculerSurface (Rectangle longueur largeur) = longueur*largeur
```

- utilisation de la fonction :

```
s1 = calculerSurface monCarre
s2 = calculerSurface monRectangle
```

Type paramétrique

- typage paramétrique :

```
renverser :: [a] -> [a]
renverser = foldl (\ acc x -> x:acc) []
```

- utilisation :

```
l1 = renverser [1..5]
l2 = renverser "hello"
```

Classes de type

- indiquer une contrainte de type :

```
doubler :: Num a => a -> a
doubler x = x*2
```

- utilisation avec différents types de la classe :

```
n1 :: Int
n1 = doubler 21

n2 :: Float
n2 = doubler 1.2
```

Classes prédéfinies

classe	fonctions de la classe	types de la classe
Eq	== /=	Int Float String...
Show	show	Int Float String...
Num	+ - * negate abs signum	Int Float...

Instances de classe

- implémenter les fonctions d'une classe pour un type :

```
instance Show Forme where
  show (Carre c) = "carre de côté " ++ (show c)
  show (Rectangle w h) = "rectangle de longueur " ++ (show w) ++ " et de largeur " ++ show h
```

- utilisation :

```
main = do
  print (Carre 12)
  print (Rectangle 12 3)
```

Définir une nouvelle classe

- définir une classe et ses fonctions à implémenter :

```
class Surfacable a where
  surface :: a -> Float
```

- utilisation :

```
instance Surfacable Forme where
  surface = calculerSurface

surfaces :: Surfacable a => [a] -> [Float]
surfaces = map surface

main = do
  print $ surfaces [Carre 12, Rectangle 12 3]
```

Entrées/sorties avec `IO`

- fonction `main` :

```
main :: IO ()
main = print 42
```

- notation `do` :

```haskell
main :: IO ()
main = do
  putStrLn "hello"
  print 42
```

- récupérer une entrée :

```haskell
main = do
  putStrLn "entrez un texte : "
  ligne <- getLine
  putStrLn ligne
```

Monades

- design pattern modélisant la notion de séquence et de composition
- implémenté par une classe définissant les opérateurs `>>=` (bind), `>>` (then) et `return`
- la notation `do` un raccourci syntaxique
- exemples de monades : `IO`, `Maybe`…

```haskell
renverser :: String -> IO String
renverser = return . reverse

-- avec la notation do
main = do
  putStrLn "entrez un texte : "
  ligne <- getLine
  engil <- renverser ligne
  putStrLn engil

-- avec les opérateurs
main = putStrLn "entrez un texte : " >> getLine >>= renverser >>= putStrLn
```

2018-01-02 Generated by Hakyll