Chapitre 3 : Les bases de Haskell, partie 2

Carole Porrier
carole.porrier@univ-paris13.fr

Département d'informatique
IUT de Villetaneuse

28 mars 2024
Paradigmes de développement universels

Table des matières

Table des matières

Exercice

- Implémentons un jeu de **tic-tac-toe** (aussi appelé **morpion**);
- Nous aimerions pouvoir **jouer une partie**...
- ... et savoir qui **a gagné**, ou si la partie est **nulle**.
- Idéalement, il faudrait **détecter** les coups illégaux.

```
-- Une case est soit occupée par X, par O ou vide.
data Case = X | O | Vide
-- Une grille de cases 3 x 3
type Grille = ((Case, Case, Case),
               (Case, Case, Case),
               (Case, Case, Case))
-- Grille résultante après une suite de coups valides
jouerPartie :: [(Int,Int)] -> Grille
-- Résultats possibles d'une partie
data EtatPartie = EnCours | XGagne | OGagne | Nulle
-- Résultat d'une partie
etatGrille :: Grille -> EtatPartie
```

Table des matières

Retour sur le filtrage

- Un raccourci pratique lorsqu'on utilise le **filtrage** (*pattern matching*): le caractère **arobase** (@).
- Permet d'utiliser le **motif** complet.

```
headTail :: String -> String
headTail ""         = ""
headTail s@(x:xs) = "La premiere lettre de " ++ s ++
                    " est '" ++ [x] ++
                    "' et sa queue est " ++ xs
```

```
Prelude> headTail "Bonjour"
"La premiere lettre de Bonjour est 'B' et sa queue est
    onjour"
```

Des fonctions partout

- Une fonction peut avoir une autre fonction en **paramètre**;
- Une fonction peut **retourner** une fonction;
- Les fonctions peuvent être **composées** (lorsque **compatibles**);
- Les fonctions peuvent être appliquées **partiellement**.
- Exemple **typique**: `map` et `filter`.

Curryfication

- Le processus qui permet d'appliquer **partiellement** des fonctions est appelé **curryfication**;
- En Haskell, l'associativité est de **gauche à droite** par défaut:
- Par exemple, considérons les **signatures** suivantes:

```
max :: Ord a => a -> a -> a     -- La fonction max
max :: Ord a => a -> (a -> a)   -- Équivalent!
max :: Ord a => (a,a) -> a      -- Pas équivalent
```

- Autrement dit, `max` est une fonction qui prend **une valeur** et qui produit une fonction de type `Ord a =>a -> a`.
- Il y a donc un **intérêt** à distinguer les fonctions utilisant des **tuples** par rapport à celles qui n'en utilisent pas!

Exemple

- Une fonction très **pratique**: `zipWith`;
- Peut être utilisée sur une **partie** de la **même** liste!

```
*Main> zipWith (+) [1,2,3] [4,5,6]
[5,7,9]
*Main> zipWith max [1,2,3] [4,5,6]
[4,5,6]
*Main> zipWith ((++) . (++" ")) ["je", "tu", "il"] ["suis", "es"
    , "est"]
["je suis","tu es","il est"]
*Main> let l = [1,4,8,9,11]
*Main> zipWith (-) (drop 1 l) l
[3,4,1,2]
```

La fonction `flip`

- Une autre fonction très **pratique**, dont l'implémentation est équivalente à celle-ci

```
flip :: (a -> b -> c) -> b -> a -> c
flip f y x = f x y
-- Si on ajoute les parentheses, on voit mieux!
flip :: (a -> b -> c) -> (b -> a -> c)
```

- Elle permet d'**inverser** l'ordre des paramètres d'une fonction;

- Exemples :

```
*Main> (-) 9 4
5
*Main> (flip (-)) 9 4
-5
*Main> (flip (:)) [1,2,3] 0
[0,1,2,3]
```

Fonctions anonymes (lambdas)

- Parfois, il est plus facile d'écrire une fonction **anonyme**;
- Syntaxe: `\variables -> expression`.
- **Attention!** N'utiliser cette syntaxe que lorsqu'il n'y a pas mieux...

```
-- Préférer cette forme
*Main> map (+3) [1,3,5,6,7]
[4,6,8,9,10]
-- Plutot que celle-ci
*Main> map (\x -> x + 3) [1,3,5,6,7]
[4,6,8,9,10]
-- Dans ce cas, comment faire?
*Main> filter (\(x,y) -> x /= y) [(1,1),(1,2),(2,2),(2,3)]
[(1,2),(2,3)]
```

- Définition de `flip` avec lambda:

```
flip :: (a -> b -> c) -> b -> a -> c
flip f = \x y -> f y x
```

Repliements (`fold`)

- Patron **fréquent**: **parcours** d'un conteneur pour calculer un **résutat**;
- Deux fonctions très **pratiques**: `foldl` et `foldr`:

```
*Main> :t foldl
foldl :: (b -> a -> b) -> b -> [a] -> b
*Main> :t foldr
foldr :: (a -> b -> b) -> b -> [a] -> b
*Main> foldr (+) 0 [1,2,3,4,5]
15
*Main> foldl (++) [] ["un ", "deux ", "trois"]
"un deux trois"
```

- Si on sait que la liste n'est **pas vide**, on peut utiliser `foldl1` et `foldr1`:

```
*Main> foldr1 (+) [1,2,3,4,5]
15
*Main> foldl1 (++) ["un ", "deux ", "trois"]
"un deux trois"
```

Exemple

- La fonction `foldl` “replie” une liste de la gauche vers la droite:

```
*Main> :t foldl
foldl :: (b -> a -> b) -> b -> [a] -> b
```

- La fonction `reverse` en Haskell est implémentée comme suit:

```
reverse :: [a] -> [a]
reverse = foldl (flip (:)) []

-- reverse "jour" = foldl (flip (:)) [] "jour"
--                = 'r':'u':'o':'j':[]
```

Les fonctions `scanr` et `scanl`

- Les **repliements** servent à produire **un seul résultat** après avoir parcouru une collection;
- Lorsqu'on s'intéresse aux **résultats intermédiaires**, il suffit d'utiliser **`scanr`**, **`scanl`**, **`scanr1`** et **`scanl1`**.

```
*Main> :t scanr
scanr :: (a -> b -> b) -> b -> [a] -> [b]
*Main> :t scanl
scanl :: (b -> a -> b) -> b -> [a] -> [b]
*Main> scanl (+) 0 [1,2,3,4,5]
[0,1,3,6,10,15]
*Main> scanr (+) 0 [1,2,3,4,5]
[15,14,12,9,5,0]
```

Lecture et crédit

- Cours d'**aujourd'hui** et **cours précédent**: sections 3, 4, 5 et 6 de Learn You a Haskell for Great Good!.

Cours adapté de celui d'Alexandre Blondin Massé (UQAM), avec son aimable autorisation.