Fonctions d'ordre supérieur prédéfinies

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

Fonctions prédéfinies

Plusieurs fonctions de premier ordre sont déjà définies car ultra-classiques et très utiles.
Nous allons en découvrir quelques unes par la pratique.

Pour chacune des fonctionnelles suivantes, donner leur définition et un exemple d'utilisation.

Map

`map f list`

Applique `f` à tous les éléments de la liste `list`.

Map

`map f list`

Applique `f` à tous les éléments de la liste `list`.

```
map' :: (a -> b) -> [ a ] -> [ b ]
map' _ [] = []
map' f (h:t) = f h:(map' f t)
```

Filter

`filter predicat list`

Supprime tous les éléments de la liste `list` qui ne vérifie pas le prédicat.

Filter

`filter predicat list`

Supprime tous les éléments de la liste `list` qui ne vérifie pas le prédicat.

```
filter' :: (a -> Bool) -> [ a ] -> [ a ]
filter' _ [] = []
filter' pred (h:t) | pred h = h:(filter' pred t)
                   | otherwise = filter' pred t
```

Dropwhile

`dropwhile predicat list`

Supprime tous les éléments en tête de la liste `list` tant que le prédicat est vérifié.

Dropwhile

`dropwhile predicat list`

Supprime tous les éléments en tête de la liste `list` tant que le prédicat est vérifié.

```
dropWhile' :: (a -> Bool) -> [ a ] -> [ a ]
dropWhile' _ [] = []
dropWhile' pred (h:t) | pred h = dropWhile' pred t
                      | otherwise = h:t
```

Partition

`partition predicat list`

Produit 2 listes l'une vérifiant le prédicat, l'autre ne le vérifiant pas.

Partition

`partition predicat list`

Produit 2 listes l'une vérifiant le prédicat, l'autre ne le vérifiant pas.

```
partition' :: (a -> Bool) -> [ a ] -> ([ a ], [ a ])
partition' _ [] = ([], [])
partition' pred (h:t) | pred h = (h:l1, l2)
                      | otherwise = (l1, h:l2)
    where (l1, l2) = partition' pred t
```

All

`all predicat list`

Renvoie `true` lorsque le prédicat est vérifié pour tous les éléments.

All

`all predicat list`

Renvoie `true` lorsque le prédicat est vérifié pour tous les éléments.

```
all' :: (a -> Bool) -> [ a ] -> Bool
all' _ [] = True
all' pred (h:t) = pred h && (all' pred t)
```

Any

`any predicat list`

Renvoie `true` lorsque le prédicat est vérifié pour l'un des éléments.

Any

`any predicat list`

Renvoie `true` lorsque le prédicat est vérifié pour l'un des éléments.

```
any' :: (a -> Bool) -> [ a ] -> Bool
any' _ [] = False
any' pred (h:t) = pred h || (any' pred t)
```

Fold left

`foldl f accumulateur list`

`f` est une fonction à deux variables. La fonction `f` renvoie la nouvelle valeur de l'accumulateur après l'avoir appliquée sur la tête et l'accumulateur. `foldl` applique récursivement de gauche à droite la fonction `f` aux éléments de la liste à partir de la valeur initiale donnée de l'accumulateur.
Penser à la somme, au maximum, à la concaténation, etc.

Fold left

`foldl f accumulateur list`

`f` est une fonction à deux variables. La fonction `f` renvoie la nouvelle valeur de l'accumulateur après l'avoir appliquée sur la tête et l'accumulateur. `foldl` applique récursivement de gauche à droite la fonction `f` aux éléments de la liste à partir de la valeur initiale donnée de l'accumulateur.
Penser à la somme, au maximum, à la concaténation, etc.

```
foldl' :: (b -> a -> b) -> b -> [a] -> b
foldl' _ acc [] = acc
foldl' f acc (h:t) = foldl' f (f acc h) t
```

Fold right

`foldr f accumulateur list`

`f` est une fonction à deux variables. La fonction `f` renvoie la nouvelle valeur de l'accumulateur après l'avoir appliquée sur la tête et l'accumulateur. `foldr` applique récursivement de droite à gauche la fonction `f` aux éléments de la liste à partir de la valeur initiale donnée de l'accumulateur.

Fold right

`foldr f accumulateur list`

`f` est une fonction à deux variables. La fonction `f` renvoie la nouvelle valeur de l'accumulateur après l'avoir appliquée sur la tête et l'accumulateur. `foldr` applique récursivement de droite à gauche la fonction `f` aux éléments de la liste à partir de la valeur initiale donnée de l'accumulateur.

```
foldr' :: (a -> b -> b) -> b -> [a] -> b
foldr' _ acc [] = acc
foldr' f acc (h:t) = f h (foldr' f acc t)
```