Cycle ingénieur - 2ème année

Programmation fonctionnelle

Listes

2023-2024

Plan

Liste (module `Data.List`, documentation)

```
data [a] = [] | (:) a [a]
```

Définition

- `[]` : liste vide
- Instantiation directe avec les valeurs séparées par `,`
- Listes numériques à progression arithmétique avec `..`

Constructeur `(:)`

- Notation usuelle infixe : *élément* `:` *liste*
- Imbrication possible

Type récursif ⇒ filtrage par motifs uniquement

- Règles inchangées : exhaustivité, séquentialité, unicité

Liste

Expression	Résultat
`(:) 1 []`	`[1]`
`1 : []`	`[1]`
`5 : 1 : 2 : 1 : []`	`[5, 1, 2, 1]`
`[2..5]`	`[2, 3, 4, 5]`
`[2, 4..9]`	`[2, 4, 6, 8]`

```
elementCount (_ : t) = 1 + elementCount t
```

⇒ Pattern match(es) non-exhaustive

Expression	Résultat
`elementCount [5, 1, 2, 1]`	Non-exhaustive patterns

```
elementCount []      = 0
elementCount (_ : t) = 1 + elementCount t
```

Expression	Résultat
`elementCount [5, 1, 2, 1]`	4

Fonctions de base

Fonctions de base : accès

- `null :: [a] -> Bool`
  `null l` indique si la liste est vide
- `head :: [a] -> a`
  `head l` renvoie la « tête » de `l`, i.e. `head (h : t) = h`
- `tail :: [a] -> [a]`
  `tail l` renvoie la « queue » de `l`, i.e. `tail (h : t) = t`
- `init :: [a] -> [a]`
  `init l` renvoie la liste `l` ôtée de son dernier élément
- `last :: [a] -> a`
  `last l` renvoie le dernier élément de `l`,
- `length :: [a] -> Int`
  `length l` renvoie le nombre d'éléments de `l`
- `(!!) :: [a] -> Int -> a`
  `l !! n` renvoie l'élément de `l` en position `n` (`0` pour la tête)

Fonctions de base : accès

Expression	Résultat
`null []`	`True`
`null [5, 1, 2, 1]`	`False`
`head [5, 1, 2, 1]`	`5`
`head []`	`Empty list`
`tail [5, 1, 2, 1]`	`[1, 2, 1]`
`tail []`	`Empty list`
`init [5, 1, 2, 1]`	`[5, 1, 2]`
`init []`	`Empty list`
`last [5, 1, 2, 1]`	`1`
`last []`	`Empty list`
`length [5, 1, 2, 1]`	`4`
`[5, 1, 2, 1] !! 0`	`5`
`[5, 1, 2, 1] !! 4`	`Index too large`
`[5, 1, 2, 1] !! (-1)`	`Negative index`

Fonctions de base : autres

- `elem : Eq a => a -> [a] -> Bool`
  `elem x l` vérifie si `x` est un élément de `l`
  Cette fonction est souvent utilisée de manière infixe.
- `notElem = not . elem`
- `lookup : Eq a => a -> [(a, b)] -> Maybe b`
  `lookup k l` renvoie :
  - `Just y` où `(x, y)` est le premier couple tel que `x == k`
  - `Nothing` si un tel couple n'existe pas
- `reverse : [a] -> [a]`
  `reverse l` renvoie la liste « miroir » de `l`
- `(++) : [a] -> [a] -> [a]`
  `l1 ++ l2` renvoie la concaténation de `l1` puis `l2`
- `concat : [[a]] -> [a]`
  `concat ll` renvoie la concaténation de toutes les listes contenues dans `ll`.

Fonctions de base : autres

Expression	Résultat
`elem 2 [5, 1, 2, 1]`	`True`
`` 0 `elem` [5, 1, 2, 1] ``	`False`
`lookup 2 [(5, 4), (2, 1)]`	`Just 1`
`lookup 0 [(5, 4), (2, 1)]`	`Nothing`
`reverse []`	`[]`
`reverse [5, 1, 2, 1]`	`[1, 2, 1, 5]`
`[5, 1, 2] ++ [1, 4]`	`[5, 1, 2, 1, 4]`
`concat [[5, 1, 2], [1], [4]]`	`[5, 1, 2, 1, 4]`

Fonctions à prédicat

Fonctions à prédicat : recherche

Prédicat ≡ `a -> Bool`

- Fonction de test sur le type des éléments de la liste

Fonctions à prédicat : recherche

- `any : (a -> Bool) -> [a] -> Bool`
  `any p l` vérifie si il existe un élément `x` de `l` tel que `p x = True`
- `all : (a -> Bool) -> [a] -> Bool`
  `all p l` vérifie si `p` donne `True` pour tous les éléments de `l`
- `find : (a -> Bool) -> [a] -> Maybe a`
  `find p l` renvoie :
  - `Just x` où `x` est le premier élément de `l` tel que `p x = True` (si il existe)
  - `Nothing` sinon
- `findIndex : (a -> Bool) -> [a] -> Maybe Int`
  renvoie l'indice du premier élément (si il existe) au lieu de l'élément lui-même

Fonctions à prédicat : recherche

Expression	Résultat
`any (> 1) [5, 1, 2, 1]`	`True`
`all (> 1) [5, 1, 2, 1]`	`False`
`find (> 1) [5, 1, 2, 1]`	`Just 5`
`find (< 1) [5, 1, 2, 1]`	`Nothing`
`findIndex (> 1) [5, 1, 2, 1]`	`Just 0`
`findIndex (< 1) [5, 1, 2, 1]`	`Nothing`

Fonctions à prédicat : filtrage

- `filter : (a -> Bool) -> [a] -> [a]`
  `filter p l` renvoie la liste des éléments *x* de `l` pour lesquels `p` *x* = `true` *en conservant l'ordre initial des éléments*

- `partition : (a -> Bool) -> [a] -> ([a], [a])`
  `partition p l` renvoie un couple de deux listes :
  - le résultat de `filter p l`
  - le résultat de `filter (not . p) l`

  (optimisé par rapport à deux appels séparés de `filter`)

Expression	**Résultat**
`filter (> 1) [5, 1, 2, 1]`	`[5, 2]`
`partition (> 1) [5, 1, 2, 1]`	`([5, 2], [1, 1])`

Fonctions d'extraction

Fonctions d'extraction inconditionnelle

- `take :: Int -> [a] -> [a]`
  `take n l` renvoie les `n` premiers éléments de `l`
- `drop :: Int -> [a] -> [a]`
  `drop n l` renvoie `l` sans ses `n` premiers éléments
- `splitAt :: Int -> [a] -> ([a], [a])`
  `splitAt n l ≡ (take n l, drop n l)`
  optimisé par rapport à deux appels séparés de `take` et `drop`

Expression	**Résultat**
`take 2 [5, 1, 2, 1]`	`[5, 1]`
`drop 2 [5, 1, 2, 1]`	`[2, 1]`
`splitAt 2 [5, 1, 2, 1]`	`([5, 1], [2, 1])`

Fonctions d'extraction conditionnelle

- `takeWhile :: (a -> Bool) -> [a] -> [a]`
  `takeWhile p l` renvoie la sous-liste de `l` des éléments jusqu'au premier élément (exclus) *x* tel que `p` *x* `== False`
- `dropWhile :: (a -> Bool) -> [a] -> [a]`
  `dropWhile p l` renvoie la sous-liste de `l` des éléments commençant par le premier élément *x* tel que `p` *x* `== False`
- `span :: (a -> Bool) -> [a] -> ([a], [a])`
  `span p l ≡ (takeWhile p l, dropWhile p l)`
  optimisé par rapport à deux appels séparés de `takeWhile` et `dropWhile`

Expression	**Résultat**
`takeWhile (>= 2) [5, 1, 2, 1]`	`[5]`
`dropWhile (>= 2) [5, 1, 2, 1]`	`[1, 2, 1]`
`span (>= 2) [5, 1, 2, 1]`	`([5], [1, 2, 1])`

Fonctions de transformation et d'agrégation

Fonctions de transformation

- `map : (a -> b) -> [a] -> [b]`
  `map f l` renvoie la liste des valeurs `f` *x* pour tout élément *x* de `l` *en conservant l'ordre initial des éléments*
- `concatMap : (a -> [b]) -> [a] -> [b]`
  `concatMap f l ≡ concat (map f l)` :
  1. applique `f` à tous les éléments de `l` : chaque valeur est une *liste*
  2. concatène tous les listes du résultat précédent
- `mapMaybe : (a -> Maybe b) -> [a] -> [b]`
  Fait partie du module Data.Maybe
  `mapMaybe f l` :
  1. applique `f` à tous les éléments de `l`, i.e. `map f l`
  2. enlève tous les résultats `Nothing` de la liste obtenue
  3. extrait les résultats des constructeurs `Just`

Fonctions de transformation

Traduction de mapMaybe

```
mapMaybe f l = map fromJust $ filter isJust $ map f l
mapMaybe f l = map f l & filter isJust & map fromJust
```

Exemples

- `map (1 /) [5, 1, 2, 1]`
  ⟶ `[0.2, 1., 0.5, 1.]`
- `concatMap (\x -> [x, x]) [5, 1, 2, 1]`
  ⟶ `[5, 5, 1, 1, 2, 2, 1, 1]`
- `mapMaybe (\x -> find (== x) [2, 3]) [5, 1, 2]`
  ⟶ `[2]`

Méthodes de transformations : listes en compréhension

Formes de base

- `[f(x) | x <- l]` ⟺ `map f l`
- `[f(x) | x <- l, p x]` ⟺ `map f $ filter p $ l`

Combinaison des formes

- `[f x1 x2 | x1 <- l1, x2 <- l2]`
  ⟺ `concatMap (\x1 -> map (\x2 -> f x1 x2) l2) l1`

Expression	Résultat
`[1 / x | x <- [5, 1, 2]]`	`[0.2, 1.0, 0.5]`
`[1 / x | x <- [5, 1, 2], x > 1]`	`[0.2, 0.5]`
`[x + y | x <- [5, 1, 2], y <- [2, 3]]`	`[7, 8, 3, 4, 4, 5]`

Méthodes de transformations : listes en compréhension

Avantages

- Filtrage optimisé
- Meilleure lisibilité pour des calculs complexes

```haskell
couplesWithSum n =
  filter (\(i, j) -> i + j == n) $
    concatMap (\i -> map (\j -> (i, j)) [0..n]) [0..n]

couplesWithSum n =
   [(i, j) | i <- [0..n], j <- [0..n], i + j == n]
```

Expression	Résultat
`couplesWithSum 4`	`[(0, 4), (1, 3), (2, 2), (3, 1), (4, 0)]`

Fonctions d'agrégation : `foldl`

```
foldl :: (b -> a -> b) -> b -> [a] -> b
```

- ```
  foldl _  z []       = z
  foldl op z (h : t) = foldl op (op z h) t
  ```
  $\Longrightarrow$ **récursive terminale**
- `foldl op z l` :

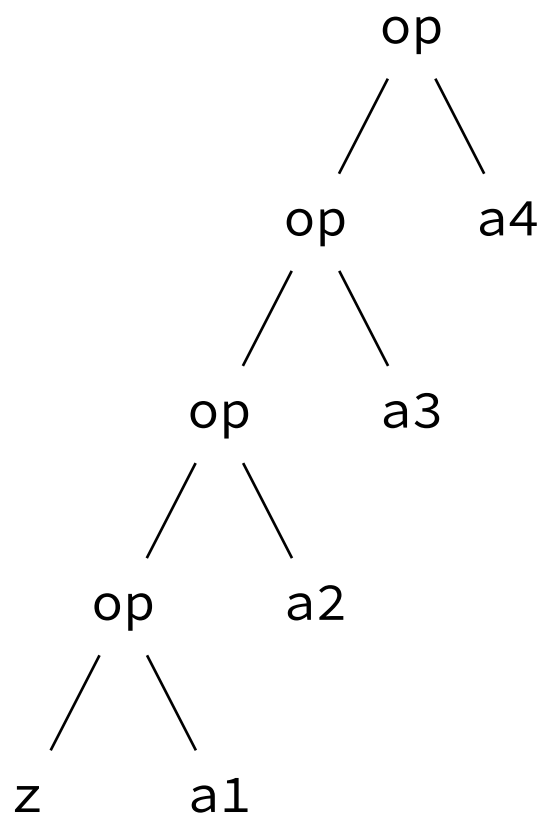
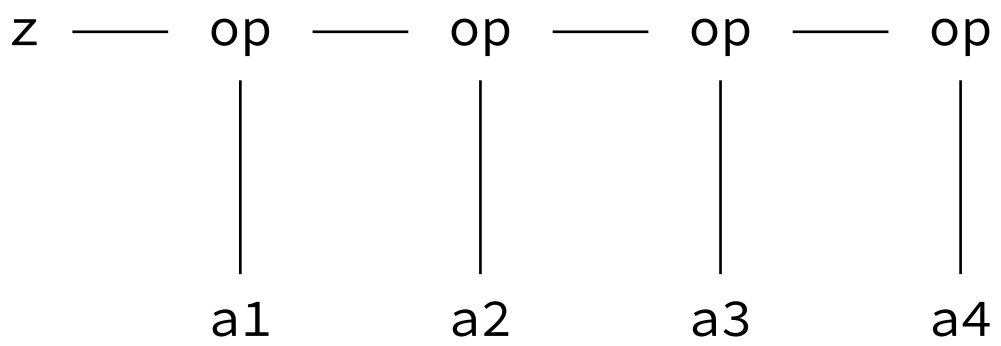
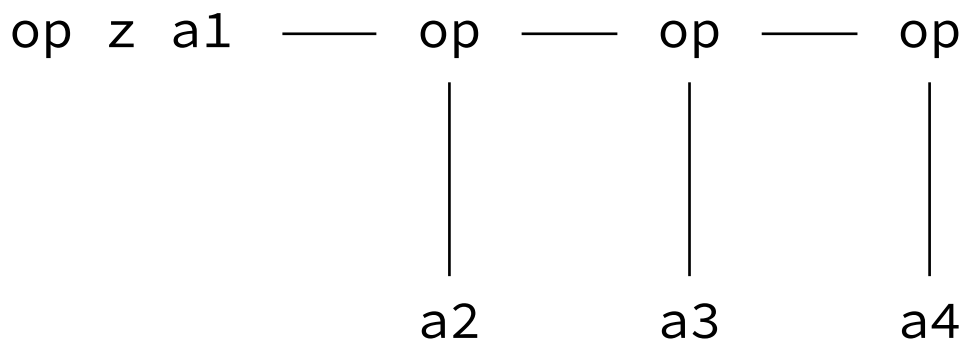
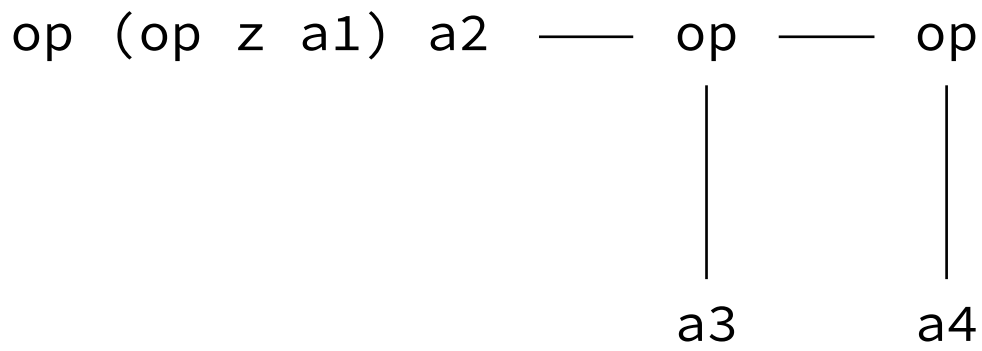
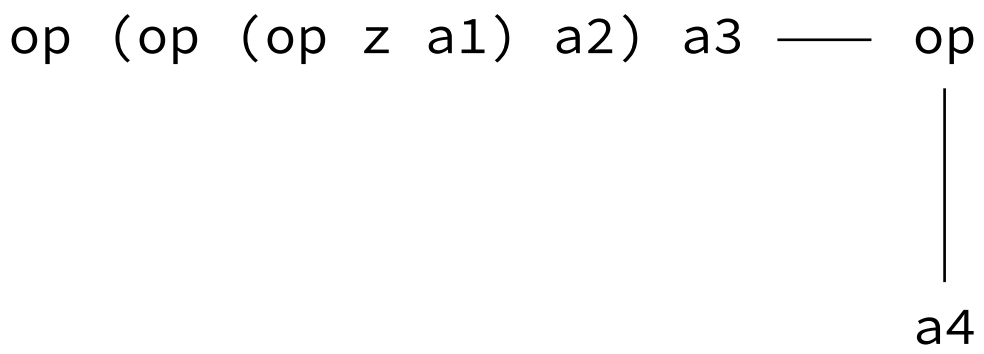
  res ← `z`

  pour tout *x* dans `l`

  res ← `op` *res x*

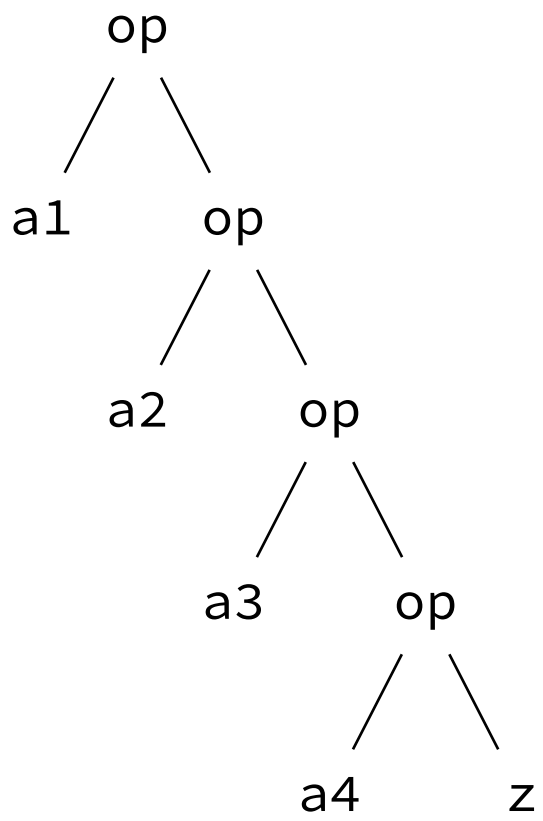
  fin pour

  retourner *res*
- `foldl op z [a1, a2, ..., an]` calcule

  `op (... (op (op z a1) a2) ...) an`
  ```

Fonctions d'agrégation : `foldl`

Exemple sur une liste de quatre éléments



Processus de réduction de `foldl`

```
foldl op z [a1, a2, a3, a4]
```



Processus de réduction de `foldl`

`foldl op z [a1, a2, a3, a4]`



Processus de réduction de `foldl`

`foldl op z [a1, a2, a3, a4]`



Processus de réduction de `foldl`

`foldl op z [a1, a2, a3, a4]`



Pourquoi « `l` » (*left*) ?

- Résultat : paramètre de *gauche* de `op`
- Éléments de la liste traités en commençant par la *gauche*
- Réduction de l'arbre par la *gauche*

Fonctions d'agrégation : `foldl`

Identifier les fonctions prédéfinies suivantes :

```
f1 = foldl (\z x -> z + 1) 0
```

```
f2 = foldl (flip (:)) []
```

Fonctions d'agrégation : `foldr`

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

- ```
  foldr _  z []       = z
  foldr op z (h : t) = op h (fold_right op t z)
  ```
  ⟹ **récursive non terminale**
- `foldr op z l` :

  res ← z

  pour tout *x* dans `reverse l`

  res ← op *x res*

  fin pour

  retourner *res*
- `foldr op z [a1, a2, ..., an]` calcule

  `op a1 (op a2 (... (op an z) ...))`
  ```

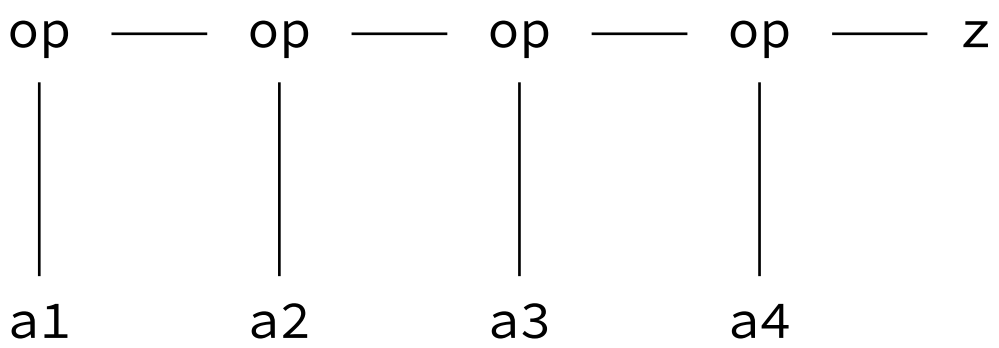
Fonctions d'agrégation : `foldr`

Exemple sur une liste de quatre éléments



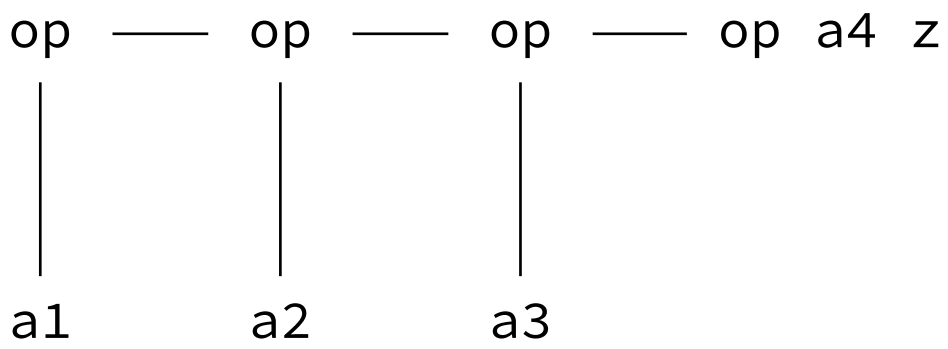
Processus de réduction de `foldr`

```
foldr op z [a1, a2, a3, a4]
```



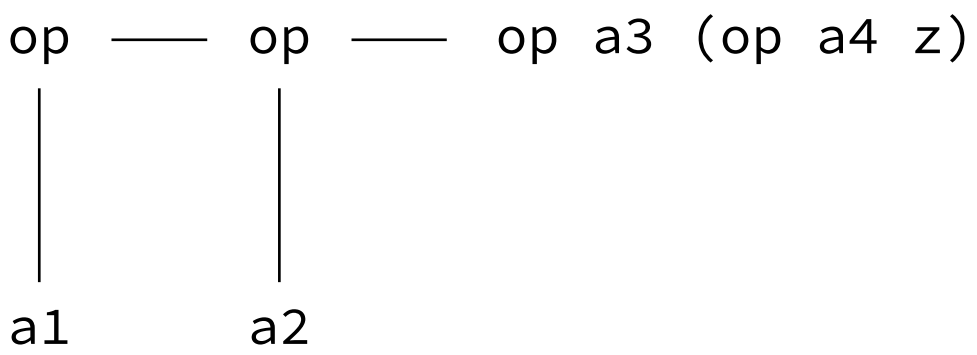
Processus de réduction de `foldr`

```
foldr op z [a1, a2, a3, a4]
```



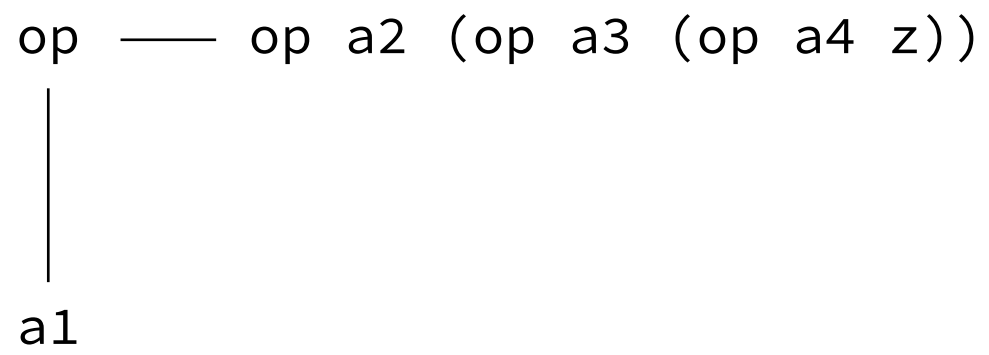
Processus de réduction de `foldr`

`foldr op z [a1, a2, a3, a4]`



Processus de réduction de `foldr`

`foldr op z [a1, a2, a3, a4]`



Pourquoi « r » (*right*) ?

- Résultat : paramètre de *droite* de op
- Éléments de la liste traités en commençant par la *droite*
- Réduction de l'arbre par la *droite*

Fonctions d'agrégation : `foldl`

Identifier la fonction prédéfinie suivante :

```
f3 = flip (foldr (:))
```

`foldl` ou `foldr` ?

Rôle des paramètres

- z est la valeur de départ *et le résultat si `l` est vide*
- op décrit la mise en à jour en fonction de l'élement courant

Aucune différence théorique...

```
foldr op z l ≡ foldl (flip op) z (reverse l)
```

Autrement dit,

- en inversant l'ordre des paramètres de op ;
- en considérant la liste « miroir » ;

on passe d'une fonction à l'autre

... mais `foldl` est généralement préférable

- `foldl` est **récursive terminale**.
- `foldr` ne l'est pas.

Fonctions d'agrégation cumulative

- `scanl :: (b -> a -> b) -> b -> [a] -> [b]`
  `scanl op z l` calcule la liste de tous les résultats intermédiaires de `foldl op z l`
  - `head (scanl op z l) = z`
  - `last (scanl op z l) = foldl op z l`
- `scanr :: (a -> b -> b) -> b -> [a] -> [b]`
  `scanr op z l` calcule la liste de tous les résultats intermédiaires de `foldr op z l`
  - `head (scanr op z l) = foldr op z l`
  - `last (scanr op z l) = z`

Expression	**Résultat**
`scanl (+) 0 [5, 1, 2, 1]`	`[0, 5, 6, 8, 9]`
`scanr (+) 0 [5, 1, 2, 1]`	`[9, 4, 3, 1, 0]`

Intérêt des fonctions du module `List`

Ne plus écrire de fonctions récursives

- Récursivité « incluse » dans les fonctions elles-même

On se concentre sur le cas général...

- ... et non plus sur le cas terminal
- Exemple : `filter`, `find`, etc...

Choix réduit (et donc simplifié) du type de traitement

- `any` / `all` ou `find`
- `filter` / `partition`
- `map` / `concatMap` / `mapMaybe`
- `foldl` / `foldr` ou `scanl` / `scanr`

Autres fonctions

Autres fonctions : listes de couples

- `unzip :: [(a, b)] -> ([a], [b])`
  `unzip l ≡ (map fst l, map snd l)`
  `unzip l` sépare la liste `l` de couples en deux listes :
  - la liste des premières composantes,
  - la liste des secondes composantes.
- `zip :: [a] -> [b] -> [(a, b)]`
  `zip` est l'opération inverse de `unzip` : `zip (unzip l) = l`
  `zip l1 l2` renvoie la liste des couples dont :
  - les premières composantes sont dans `l1`
  - les secondes composantes sont dans `l2`

  Les éléments surnuméraires ne sont pas pris en compte.

Il existe des variations pour les listes de 3-uplets à 7-uplets.

Autres fonctions : listes de couples

Exemples

- `unzip [(1, 'a'), (2, 'b'), (3, 'c')]`
  ⟶ `([1, 2, 3], ['a', 'b', 'c'])`
- `zip [1, 2, 3] ['a', 'b', 'c']`
  ⟶ `[(1, 'a'), (2, 'b'), (3, 'c')]`
- `zip [1, 2] ['a', 'b', 'c']`
  ⟶ `[(1, 'a'), (2, 'b')]`

Autres fonctions : tri

- `sort : Ord a => [a] -> [a]`
  `sort l` trie `l` selon l'ordre défini par `(<=)` *en conservant l'ordre initial des éléments égaux entre eux*
- `sortOn : Ord b => (a -> b) -> [a] -> [a]`
  `sort f l` trie `l` selon les valeurs de `map f l` *en conservant l'ordre initial des éléments égaux entre eux*

Exemples

- `sort [5, 1, 2, 1]`
  ⟶ `[1, 1, 2, 5]`
- `sortOn negate [5, 1, 2, 1]`
  ⟶ `[5, 2, 1, 1]`
- `sortOn length [[2..5], [1..2], [1..4], [1..2]]`
  ⟶ `[[1, 2], [1, 2], [2, 3, 4, 5], [1, 2, 3, 4]]`

… et il y en d'autres !

La liste n'est pas exhaustive !

- variantes de certaines fonctions (comme `map`) avec deux listes
- …

Consulter la documentation officielle

Une chaîne de caractères est une liste de caractères

`type String = [Char]` (*Documentation officielle du type `String`*)

Toute fonction applicable sur une liste est applicable sur une chaîne de caractères.

Il existe aussi d'autres structures de données

- `Data.Map` : tableaux associatifs
- `Data.Set` : ensembles (au sens mathématique)
- `Data.Tree` : arbres d'arité quelconque

Listes « infinies »

Évaluation stricte *vs* non-stricte

```
g :: Int -> Int -> Int
g x y = y + 1
f :: Int -> Int -> Int -> Int
f a b c = g (a * 1000) c
```

Évaluation stricte

On évalue les paramètres avant d'appliquer la fonction.

1. `f 1 (1234 * 1234) 2`
2. `f 1 1522756 2`
3. `g (1 * 1000) 2`
4. `g 1000 2`
5. `2 + 1`
6. `3`

Évaluation non-stricte

On applique la fonction avant d'évaluer les paramètres.

1. `f 1 (1234 * 1234) 2`
2. `g (1 * 1000) 2`
3. `2 + 1`
4. `3`

(Source : haskell.mooc.fi)

Les fonctions sont pures, donc le résultat est identique.

Évaluation stricte *vs* non-stricte

Évaluation non-stricte

- **Mode d'évaluation par défaut en Haskell**
- Seules les expressions nécessaires sont évaluées
  $\Longrightarrow$ évaluation « paresseuse » (cf. opérateurs booléens)

Évaluation stricte

- Plus facile à implémenter et plus économique
  $\Longrightarrow$ utilisée par la très grande majorité des langages
- Certains de ces langages (souvent fonctionnels) proposent des constructions supplémentaires d'évaluation « paresseuse » (par exemple OCaML ou Scala)

Liste « infinie »

Liste en Haskell

- La tête et la queue sont évaluées de manière non-stricte.

⇒ Production des éléments uniquement à la demande

⇒ *Possibilité de créer des séquences infinies (conceptuellement)*

Exemple : `[0..]` est la liste des entiers naturels.

Support des listes « infinies »

- **Complet** : fonctions terminant pour **toutes** les séquences infinies parce qu'elles ne forcent pas l'évaluation *(transformation, filtrage)*
- **Partiel** : fonctions terminant pour *certaines* séquences infinies *(recherche)*
- **Impossible** : fonctions ne terminant pour aucune séquence infinie *(agrégation)*

Support des listes « infinies »

Fonctions de base

- `null`, `head`, `tail`, `init`, `last`, `length`, `(!!)`
- `elem`, `reverse`, `(++)`, `concat`

Fonctions à prédicat

- `any`, `all`, `find`, `findIndex`
- `filter`, `partition`

Fonctions d'extraction

- `take`, `drop`, `splitAt`
- `takeWhile`, `dropWhile`, `span`

Fonctions de transformation

- `map`, `concatMap`, `mapMaybe`

  $\Longrightarrow$ *listes en compréhension*

Fonctions d'agrégation

- `foldl`, `foldr`, `scanl`, `scanr`

Autres fonctions

- `zip`, `unzip`, `sort`, `sortOn`

Fonction de construction pas-à-pas

```
unfoldr :: (b -> Maybe (a, b)) -> b -> [a]
unfoldr f x = case f x of
    | Nothing      -> []
    | Just (h, x1) -> h : (unfoldr f x1)
```

`unfold f x` renvoie

- la séquence vide si `f x` renvoie `Nothing`,
- la séquence avec :
  - h pour tête
  - `unfoldr f x1` comme queue

  si `f x` renvoie `Just (h, x1)`

Ne fait aucune évaluation

Fonctions de construction de séquences infinies

- `cycle : [a] -> [a]`
  `cycle s` construit la séquence obtenue en répétant `s` indéfiniment
- `repeat : a -> [a]`
  `repeat x` ≡ `cycle [x]`
  `repeat x` construit la séquence répétant l'élément `x` indéfiniment
- `iterate : (a -> a) -> a -> [a]`
  `iterate f x` construit la séquence :
  - dont la tête est `x`,
  - puis dont chaque élément est obtenu en appliquant `f` au précédent.

Listes infinies : exemples

Liste infinie des entiers naturels

```
nats = [0..]
nats = iterate (+ 1) 0
nats = unfold (\n -> Just (n, n + 1)) 0
```

Liste infinie des entiers naturels pairs

```
evens = [0, 2..]
evens = iterate (+ 2) 0
evens = unfold (\n -> Just (n, n + 2)) 0
evens = filter (\n -> n `mod` 2 == 0) nats
evens = map (* 2) nats
```

Crible d'Érathostène

```
sieve [] = []
sieve (n : ns) = n : sieve [m | m <- ns, m `mod` n /= 0]
primes = sieve [2..]
```

Expression	Résultat
`take 10 primes`	`[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]`