

# Haskell (IR3) – Listes

Fabian Reiter (fabian.reiter@univ-eiffel.fr)  
Stéphane Vialette (stephane.vialette@univ-eiffel.fr)

25 septembre 2020

## Question 1: Expressions de liste

- (a) Une liste s'écrit entre crochets, avec les éléments de la liste séparés par des virgules. Rappelez ce que font les opérateurs `[]`, `:` et `++`.

### Solution:

- `[]` est un *data constructor* qui représente la liste vide.

```
>>> :type []
[] :: [a]
```

Notez ici que `[a]` est un *polymorphic data type*.

- `:` est également un *data constructor* qui ajoute un élément en tête de liste.

```
>>> :type (:)
(:) :: a -> [a] -> [a]
```

- `++` est une fonction qui permet de concaténer deux listes de même type.

```
>>> :type (++)
(++) :: [a] -> [a] -> [a]
>>> [1..3] ++ []
[1,2,3]
>>> [] ++ [4..6]
[4,5,6]
>>> [1..3] ++ [4..6]
[1,2,3,4,5,6]
```

- (b) Évaluez les expressions de liste suivantes :

- `1:[2]`
- `[3,4]++[1,2]`
- `[3..10]`
- `tail [1..4] ++ 5:[]`
- `head [1..4] : [5]`
- `reverse [1..4] ++ [5]`
- `1 : reverse [2..5]`

### Solution:

Ne pas oublier ici que l'application de fonction (*e.g.*, `head`, `tail`, ...) a la plus forte précedence.

```
>>> 1:[2]
[1,2]
```

```
>>> [3,4]++[1,2]
[3,4,1,2]
>>> [3..10]
[3,4,5,6,7,8,9,10]
>>> tail [1..4] ++ 5:[ ] -- = [1..4] ++ [5] = [1..5]
[2,3,4,5]
>>> head [1..4] : [5] -- 1 : [5]
[1,5]
>>> reverse [1..4] ++ [5] -- = [4,3,2,1] ++ [5]
[4,3,2,1,5]
>>> 1 : reverse [2..5] -- = 1 : [5,4..2]
[1,5,4,3,2]
```

### Question 2: Définition de fonctions sur les listes

- (a) Rappelez ce que font les fonctions `head`, `tail`, `reverse`, `length`, `drop`, `take`, `!!`, et `tails`, définie dans `prelude.hs`.

#### Solution:

- `head` prend une liste et retourne son premier élément. Cette fonction lève une exception si la liste est vide.

Une implémentation possible est la suivante :

```
head :: [a] -> a
head []      = error "*** Exception: head: empty list"
head (x : _) = x
```

- `tail` prend une liste et retourne sa queue. En d'autres termes, elle coupe la tête de la liste.

Une implémentation possible est la suivante :

```
tail :: [a] -> [a]
tail []      = error "*** Exception: tail: empty list"
tail (_ : xs) = xs
```

- `reverse` renverse une liste.
- `length` prend une liste et retourne sa longueur.
- `drop` prend un nombre et une liste. Elle jette le nombre d'éléments demandé du début de la liste.
- `take` prend un nombre et une liste. Elle extrait ce nombre d'éléments du début de la liste.
- Si vous voulez obtenir un élément d'une liste par son index, utilisez `!!`. Les indices démarrent à 0.
- `tails` prend une liste et retourne ses queues.

- (b) La fonction `last`, définie dans `prelude.hs`, sélectionne le dernier élément d'une liste. Simulez dans `ghci` le comportement de `last` exclusivement à l'aide des fonctions (i) `head` et `reverse`, (ii) `length` et `!!`, et (iii) `head`, `drop` et `length`. Donner ensuite une version récursive de la fonction `last`.

#### Solution:

- (i) A l'aide des fonctions `head` et `reverse` :

```
>>> :type head
head :: [a] -> a
>>> :type reverse
```

```
reverse :: [a] -> [a]
>>> last' xs = head (reverse xs)
>>> last' [1..5]
5
```

Par  $\eta$ -réduction et composition, on obtient :

```
>>> last' = head . reverse xs
>>> last' [1..5]
5
```

— (ii) A l'aide des fonctions `length` et `!!` :

```
>>> :type length
length :: Foldable t => t a -> Int
>>> :type (!! )
(!!) :: [a] -> Int -> a
>>> last'' xs = xs !! (length xs - 1)
>>> last'' [1..5]
5
```

— (iii) A l'aide des Fonctions `head`, `drop` et `length` :

```
>>> :type head
head :: [a] -> a
>>> :type drop
drop :: Int -> [a] -> [a]
>>> :type length
length :: Foldable t => t a -> Int
>>> last''' xs = head (drop (length xs - 1) xs)
>>> last''' [1..5]
5
```

— Une version récursive (utiliser un éditeur pour écrire la fonction et chargez la dans l'interpréteur) :

```
last' :: [a] -> a
last' [] = error "*** Exception: last: empty list"
last' [x] = x
last' (_ : xs) = last' xs

last' :: [a] -> a
last' [] = error "*** Exception: last: empty list"
last' [x] = x
last' xs = last' (tail xs)
```

Préférez la première version qui utilise le *pattern matching*.

- (c) La fonction `init`, définie dans `prelude.hs`, supprime le dernier élément d'une liste. Simulez dans `ghci` le comportement de `init` exclusivement à l'aide des fonctions (i) `take` et `length`, (ii) `tail` et `reverse` et (iii) `tails`, `reverse` et `!!`. Donner ensuite une version récursive de la fonction

### Solution:

— (i) A l'aide des fonctions `take` et `length` :

```
Prelude> :type take
take :: Int -> [a] -> [a]
Prelude> :type length
```

```
length :: Foldable t => t a -> Int
>>> init' xs = take (length xs - 1) xs
>>> init' [1..5]
[1,2,3,4]
```

— (ii) A l'aide des fonctions `tail` et `reverse` :

```
>>> init'' xs = reverse (tail (reverse xs))
>>> init'' [1..5]
[1,2,3,4]
```

Par  $\eta$ -réduction et composition, on obtient :

```
>>> init'' = reverse . tail . reverse
>>> init'' [1..5]
5
```

— (iii) A l'aide des Fonctions `tails`, `reverse` et `!!` :

```
>>> import Data.List
>>> init''' xs = reverse (tails (reverse xs) !! 1)
>>> init''' [1..5]
[1,2,3,4]
```

ou encore

```
>>> import Data.List
>>> init''' xs = let xs' = tails (reverse xs) !! 1 in reverse xs'
>>> init''' [1..5]
[1,2,3,4]
```

— Une version récursive :

```
init' :: [a] -> [a]
init' [] = error "*** Exception: init: empty list"
init' [x] = []
init' (x : xs) = x : init' xs
```

```
init' :: [a] -> [a]
init' [] = error "*** Exception: init: empty list"
init' [x] = []
init' xs = head xs : init' (tail xs)
```

Préférez la première version qui utilise le *pattern matching*.

### Question 3: Chaînes de caractères

Un *palindrome* un mot dont l'ordre des lettres reste le même qu'on le lise de gauche à droite ou de droite à gauche, comme dans la phrase "Ésope reste ici et se repose".

- (a) Comment tester si un mot (*i.e.*, une chaîne de caractères sans caractère *espace*) est un palindrome ? (Accents et majuscules ne sont pas utilisés ici.)

#### Solution:

Ce n'est pas le plus efficace (pourquoi ?), mais le plus simple est d'utiliser la fonction `reverse`.

```
>>> "" == reverse ""
True
>>> "radar" == reverse "radar"
True
>>> "toto" == reverse "toto"
```

```
False
>>>
```

- (b) Comment tester si une chaîne de caractères est un palindrome ? (Accents et majuscules ne sont pas utilisés ici mais le mot peut contenir des caractères *espace* dont il ne faut pas tenir compte.)

### Solution:

Puisque les caractères *espace* ne sont pas comptabilisés, il faut au préalable les supprimer. La fonction `removeBlanks` se charge de cette tâche.

```
>>> removeBlanks xs = [x | x <- xs, x /= ' ']
>>> :type removeBlanks -- Pourquoi ???
removeBlanks :: [Char] -> [Char]
>>> removeBlanks "riri"
"riri"
>>> removeBlanks "riri fifi"
"rirififi"
>>> removeBlanks "riri fifi et loulou"
"rirififi et loulou"
>>> removeBlanks "    " == reverse (removeBlanks "    ")
True
>>> removeBlanks "r ad a r" == reverse (removeBlanks "r ad a r")
True
>>> removeBlanks "t ot o" == reverse (removeBlanks "t ot o")
False
>>>
```

### Bonus, comment supprimer les caractères *espace* sans compréhension de liste ?

Une première version, récursive et assez classique.

```
removeBlanks :: String -> String
removeBlanks [] = []
removeBlanks (x : xs)
  | x /= ' ' = x : removeBlanks xs
  | otherwise = removeBlanks xs
```

Un pliage à droite avec une fonction *lambda*.

```
import qualified Data.Foldable as F

removeBlanks :: String -> String
removeBlanks = F.foldr (\ x acc -> if x /= ' ' then x : acc else acc) []
```

Il est bien sûr possible de remplacer la fonction *lambda* par une fonction interne (visible donc uniquement depuis `removeBlanks`).

```
removeBlanks :: String -> String
removeBlanks = F.foldr f []
  where
    f x acc
      | x /= ' ' = x : acc
      | otherwise = acc
```

L'approche la plus simple consiste à utiliser la fonctionnelle `filter` (définie dans `Data.List`) accompagnée d'un prédicat (sous forme d'une fonction *lambda* ou d'une *section*, préférez cette dernière version).

```
import qualified Data.List as L

removeBlanks :: String -> String
removeBlanks = L.filter (\ x -> x /= ' ')

removeBlanks :: String -> String
removeBlanks = L.filter (/= ' ')
```

- (c) Écrire une fonction qui teste si un mot est un palindrome ? (Accents et majuscules ne sont pas utilisés ici.) Quel doit être le type de cette fonction ?

**Solution:**

```
>>> palindrome xs = xs == reverse xs
>>> :type palindrome -- Pourquoi Eq a ???
palindrome :: Eq a => [a] -> Bool
>>> palindrome ""
True
>>> palindrome "radar"
True
>>> palindrome "toto"
False
```

- (d) Écrire une fonction qui teste si une chaîne de caractères est un palindrome ? (Accents et majuscules ne sont pas utilisés ici mais la chaîne peut contenir des caractères *espace* dont il ne faut pas tenir compte..) Quel doit être le type de cette fonction ?

**Solution:**

```
>>> palindrome' xs = removeBlanks xs == reverse (removeBlanks xs)
>>> :type palindrome' -- Où est passé Eq ???
palindrome' :: [Char] -> Bool
>>> palindrome' ""
True
>>> palindrome' "r ad a r"
True
>>> palindrome' "t ot o"
False
```

#### Question 4: Types

- (a) Quel est le type des valeurs suivantes :

```
— ['a', 'b', 'c'],
— [1, 2, 3],
— [['a', 'b'], ['c', 'd']],
— [['1', '2'], ['3', '4']],
— ('a', 'b'),
— ('a', 'b', 'c'),
— (1, 2),
— (1, 2, 3),
— [(False, '0'), (True, '1')],
```

```
— ([False, True], ['0', '1']),
— [tail, init, reverse], et
— ([tail, init, reverse], [take, drop]).
```

**Solution:**

```
>>> :type ['a', 'b', 'c']
['a', 'b', 'c'] :: [Char]

>>> :type [1, 2, 3]
[1, 2, 3] :: Num t => [t]

>>> :type [['a', 'b'], ['c', 'd']]
[['a', 'b'], ['c', 'd']] :: [[Char]]

>>> :type [['1', '2'], ['3', '4']]
[['1', '2'], ['3', '4']] :: [[Char]]

>>> :type ('a', 'b')
('a', 'b') :: (Char, Char)

>>> :type ('a', 'b', 'c')
('a', 'b', 'c') :: (Char, Char, Char)

>>> :type (1, 2)
(1, 2) :: (Num t1, Num t) => (t, t1)

>>> :type (1, 2, 3)
(1, 2, 3) :: (Num t2, Num t1, Num t) => (t, t1, t2)

>>> :type [(False, '0'), (True, '1')]
[(False, '0'), (True, '1')] :: [(Bool, Char)]

>>> :type ([False, True], ['0', '1'])
([False, True], ['0', '1']) :: ([Bool], [Char])

>>> :type [tail, init, reverse]
[tail, init, reverse] :: [[a] -> [a]]

>>> :type ([tail, init, reverse], [take, drop])
([tail, init, reverse], [take, drop])
  :: ([[a] -> [a]], [Int -> [a1] -> [a1]])
```

Notez bien qu'il est tout à fait possible (et classique) de manipuler des listes de fonctions. Cependant, par *homogénéité*, toutes les fonctions doivent avoir le même type.

**Bonus.** Les *contraintes de classe* influent-elles sur le type d'une fonction ? Quelques éléments de réponse.

```
range :: (Num a, Enum a) => a -> [a]
range n = [1..n]

range' :: (Num a, Ord a) => a -> [a]
range' n = aux 1
  where
```

```

    aux i
      | i > n      = []
      | otherwise = i : aux (i+1)

>>> :type range
range :: (Num a, Enum a) => a -> [a]
>>> :type range'
range' :: (Ord a, Num a) => a -> [a]
>>> fs = [range, range']
>>> [f 3 | f <- fs]
[[1,2,3],[1,2,3]]
>>> [range, range']

<interactive>:16:1: error:
  • No instance for (Show (Integer -> [Integer]))
    arising from a use of print
    (maybe you haven't applied a function to enough arguments?)
  • In a stmt of an interactive GHCi command: print it

```

(b) Expliquer la session suivante :

```

>>> import Data.List
>>> :type (head, take)
(head, take) :: ([a] -> a, Int -> [a] -> [a])
>>> :type [head, take]

<interactive>:1:8:
  Couldn't match type 'Int' with '[[a] -> [a]]'
  Expected type: [[a] -> [a]] -> [a] -> [a]
  Actual type: Int -> [a] -> [a]
  In the expression: take
  In the expression: [head, take]
Prelude Data.List>

```

#### Solution:

(head, take) est une paire contenant deux fonctions : la première de type `[a] -> a` et la seconde de type `Int -> [a] -> [a]`. Le type de (head, take) est donc `([a] -> a, Int -> [a] -> [a])`. Aucune difficulté ici puisque nous avons une paire.

Par contre une liste est *homogène*, elle ne peut contenir que des éléments de même type. Ce n'est à l'évidence pas le cas ici. `[head, take]`.

#### Question 5: Fonctions

Considérons les fonctions suivantes :

1. `second xs = head (tail xs)`,
2. `appl (f,x) = f x`,
3. `pair x y = (x,y)`,
4. `mult x y = x * y`,
5. `double = mult 2`,
6. `palindrome xs = reverse xs == xs`,

7. `twice f x = f (f x),`
8. `incrAll xs = map (+1) xs, et`
9. `norme xs = sqrt (sum (map f xs)) where f x = x^2.`

(a) Calculez les types de ces fonctions, en n'oubliant pas les contraintes de classe.

**Solution:**

```
>>> second xs = head (tail xs)
>>> :type second
second :: [a] -> a

>>> appl (f,x) = f x
>>> :type appl
appl :: (t1 -> t, t1) -> t

>>> pair x y = (x,y)
>>> :type pair
pair :: t -> t1 -> (t, t1)

>>> mult x y = x * y
>>> :type mult
mult :: Num a => a -> a -> a

>>> double = mult 2
>>> :type double
double :: Num a => a -> a

>>> palindrome xs = reverse xs == xs
>>> :type palindrome
palindrome :: Eq a => [a] -> Bool

>>> twice f x = f (f x)
>>> :type twice
twice :: (t -> t) -> t -> t

>>> incrAll xs = map (+1) xs
>>> :type incrAll
incrAll :: Num b => [b] -> [b]

>>> norme xs = sqrt (sum (map f xs)) where f x = x^2
>>> :type norme
norme :: Floating a => [a] -> a
>>>
```

(b) Donnez une forme currifiée de la fonction `appl'`.

**Solution:**

```
appl :: (a -> b, a) -> b
appl (f, x) = f x

appl' :: (a -> b) -> a -> b
appl' f x = f x
```

On se rappelle (!?) alors de la fonction `($)`.

```
>>> :type ($)
($) :: (a -> b) -> a -> b
```

et de fait

```
>>> appl' succ 1
2
>>> ($) succ 1
2
```

(c) Quelles sont les fonctions d'ordre supérieur ?

**Solution:**

Les fonctions Haskell peuvent prendre d'autres fonctions en paramètres et retourner des fonctions en valeur de retour. Une fonction capable d'une de ces deux choses est dite d'ordre supérieur. C'est le cas des fonctions `appl` et `twice` qui prennent une fonction en paramètre.

```
>>> increment x = x + 1
>>> :type increment
increment :: Num a => a -> a
>>> appl (increment, 1)
2
>>> appl' increment 1
2
>>> twice increment 1
3
>>>
```

(d) Quelles sont les fonctions polymorphes ?

**Solution:**

Une fonction est *polymorphe* (de la langue grecque ancienne *de plusieurs formes*) si son type contient une ou plus variables de type.

La fonction `length :: Foldable t => t a -> Int` est polymorphe tandis que `(/= 'x') :: Char -> Bool` ne l'est pas.

### Question 6: Compréhensions de listes

(a) À l'aide d'une compréhension de liste, calculer la liste de tous entiers positifs impairs.

**Solution:**

```
>>> take 10 [x | x <- [1..], odd x]
[1,3,5,7,9,11,13,15,17,19]
>>> take 10 [x | x <- [1,3..]]
[1,3,5,7,9,11,13,15,17,19]
```

(b) À l'aide d'une compréhension de liste, calculer la liste de carrés des entiers pairs (i.e., les entiers  $i^2$  pour  $i = 2, 4, \dots$ ).

**Solution:**

```
>>> take 10 [x^2 | x <- [1..], even x]
[4,16,36,64,100,144,196,256,324,400]
```

```
>>> take 10 [x^2 | x <- [2,4..]]
[4,16,36,64,100,144,196,256,324,400]
```

- (c) À l'aide d'une compréhension de liste, calculer la liste  $[[1], [1, 2], [1, 2, 3], [1, 2, 3, 4], [1, 2, 3, 4, 5]]$ .

**Solution:**

```
>>> [[1..n] | n <- [1..5]]
[[1], [1, 2], [1, 2, 3], [1, 2, 3, 4], [1, 2, 3, 4, 5]]
```

- (d) À l'aide d'une compréhension de liste, calculer la liste des paires d'entiers  $(n, m)$ ,  $1 \leq n \leq m \leq 20$  et  $\sum_i^n i = m$ .

**Solution:**

```
>>> [(m, n) | n <- [1..20], m <- [n..20], sum [i | i <- [1..n]] == m]
[(1, 1), (3, 2), (6, 3), (10, 4), (15, 5)]
```

- (e) En arithmétique, un *nombre parfait* est un entier naturel  $n$  tel que  $\sigma(n) = 2n$ , où  $\sigma(n)$  est la somme des diviseurs positifs de  $n$ . Cela revient à dire qu'un entier naturel est parfait s'il est égal à la moitié de la somme de ses diviseurs ou encore à la somme de ses diviseurs stricts. Ainsi 6 est un nombre parfait car  $2 \times 6 = 12 = 1 + 2 + 3 + 6$ , ou encore  $6 = 1 + 2 + 3$ . Les trois premiers nombres parfaits sont

—  $6 = 1 + 2 + 3$ ,

—  $28 = 1 + 2 + 4 + 7 + 14$ , et

—  $496 = 1 + 2 + 4 + 8 + 16 + 31 + 62 + 124 + 248$ .

À l'aide d'une compréhension de liste, calculer la liste des nombres parfaits (et, par exemple, donner le quatrième; indice il s'agit de 8128). Existe-t-il un nombre parfait impair ?

**Solution:**

```
>>> take 4 [x | x <- [1..],
               sum [i | i <- [1..x-1], x `mod` i == 0] == x]
[6, 28, 496, 8128]
```

- (f) En arithmétique, deux nombres entiers distincts  $n$  et  $m$  sont dits *amicaux* (ou *aimables*) si la somme des diviseurs de l'un est égale à la somme des diviseurs de l'autre et si ces deux sommes sont égales à la somme des deux nombres. Cette propriété se traduit par  $\sigma(n) = \sigma(m) = n + m$ . On exclut le cas  $n = m$ , qui correspondrait à un nombre parfait. Par exemple 220 et 284 sont amicaux car

—  $220 + 284 = 504$ ,

—  $\sigma(220) = 1 + 2 + 4 + 5 + 10 + 11 + 20 + 22 + 44 + 55 + 110 + 220 = 504$  et

—  $\sigma(284) = 1 + 2 + 4 + 71 + 142 + 284 = 504$ .

À l'aide d'une compréhension de liste, calculer la liste des paires  $(n, m)$ ,  $1\,000 \leq n < m < 1\,300$ , de nombres amicaux.

**Solution:**

```
[(n, m) | n <- [1000..1300]
          , m <- [n+1..1300]
          , sum [i | i <- [1..n], n `mod` i == 0] == n+m
          , sum [i | i <- [1..m], m `mod` i == 0] == n+m]
```

## Question 7: Fonctions simples

- (a) Écrire une fonction permettant de compter le nombre d'éléments dans une liste (sans utiliser `length` bien sûr!).

**Solution:**

Une version récursive immédiate :

```
length' :: Num a => [t] -> a
length' [] = 0
length' (_ : xs) = 1 + length' xs
```

On vérifie

```
>>> :type length'
length' :: Num a => [t] -> a
>>> length' []
0
>>> length' ['a']
1
>>> length' ['a'..'z']
26
```

Une version avec accumulateur et fonction auxiliaire :

```
length'' :: [a] -> Integer
length'' = aux 0
  where
    aux n [] = n
    aux n (_ : xs) = aux (n+1) xs
```

Bien sûr, il est possible de plier.

```
import qualified Data.Foldable as F

length''' :: (Foldable t, Num b) => t a -> b
length''' = F.foldr (\ _ acc -> acc+1) 0
```

- (b) Écrire une fonction permettant de renverser une liste (sans utiliser `reverse` bien sûr!)

**Solution:**

Une première version récursive –terriblement– inefficace à cause de l'utilisation de l'opérateur `++` :

```
reverse' [] = []
reverse' (x:xs) = reverse' xs ++ [x]
```

Une version beaucoup plus efficace (grâce, bien sûr, à l'utilisation de l'opérateur `:`)

```
reverse'' = reverseAux []
  where
    reverseAux acc [] = acc
    reverseAux acc (x:xs) = reverseAux (x:acc) xs
```

Encore mieux.

```
import qualified Data.Foldable as F

reverse''' :: Foldable t => t a -> [a]
reverse''' = F.foldl (flip (:)) []
```

- (c) Écrire une fonction permettant de calculer le nombre de voyelles dans une chaîne de caractères. (Nous ne préoccuons pas des accents).

**Solution:**

Aucune difficulté particulière mais plusieurs possibilités simples. Tout d'abord avec une compréhension de liste :

```
countVs xs = length [x | x <- xs
                        , x `elem` ['a', 'e', 'i', 'o', 'u', 'y']]
```

Avec une fonction récursive :

```
countVs' [] = 0
countVs' (x:xs)
  | v x      = 1 + countVs' xs
  | otherwise = countVs' xs
where
  v x = x `elem` ['a', 'e', 'i', 'o', 'u', 'y']
```

- (d) La fonction `splitAt` de type `Int -> [a] -> ([a], [a])` retourne un couple de listes obtenu en cassant une liste à une position donnée.

```
>>> :type splitAt
splitAt :: Int -> [a] -> ([a], [a])
>>> splitAt 0 [1..10]
([], [1,2,3,4,5,6,7,8,9,10])
>>> splitAt 5 [1..10]
([1,2,3,4,5], [6,7,8,9,10])
>>> splitAt 10 [1..10]
([1,2,3,4,5,6,7,8,9,10], [])
>>> splitAt 3 []
([], [])
>>>
```

Proposez une implémentation de `splitAt`.

**Solution:**

Une première solution qui utilise `take` et `drop` :

```
import qualified Data.List as L

splitAt' :: Int -> [a] -> ([a], [a])
splitAt' n xs = (L.take n xs, L.drop n xs)
```

Une seconde solution récursive :

```
splitAt'' :: Int -> [a] -> ([a], [a])
splitAt'' 0 xs      = ([], xs)
```

```
splitAt'' _ [] = ([], [])
splitAt'' n (x : xs)
  | n < 0      = ([], x : xs)
  | otherwise  = (x : ys, zs)
  where
    (ys, zs) = splitAt'' (n-1) xs
```

- (e) La *suite de Fibonacci* est une suite d'entiers dans laquelle chaque terme est la somme des deux termes qui le précèdent. Elle commence généralement par les termes 0 et 1 (parfois 1 et 1) et ses premiers termes sont : 0, 1, 1, 2, 3, 5, 8, 13, 21, ... (suite A000045 de l'OEIS (On-Line Encyclopedia of Integer Sequences)). Écrire une fonction `fibonacci` de type `fib :: (Num a1, Num a, Eq a) => a -> a1` permettant de calculer un terme de la suite de fibonacci.

#### Solution:

Une première version récursive peu efficace :

```
fibonacci 0 = 0
fibonacci 1 = 1
fibonacci n = fibonacci (n-1) + fibonacci (n-2)
```

En utilisant une liste infinie :

```
import qualified Data.List as L

fibonacci' :: Num a => Int -> a
fibonacci' n = L.head . L.drop n $ aux
  where
    aux = 0 : 1 : next aux
    next (a : t@(b : _)) = (a+b) : next t
```

ou encore de façon élégante :

```
import qualified Data.List as L

fibonacci'' :: Num a => Int -> a
fibonacci'' n = L.head . L.drop n $ aux
  where
    aux = 0 : 1 : L.zipWith (+) aux (L.tail aux)
```

- (f) Écrire les fonctions `oddElements` et `evenElements` qui retournent les listes constituées des éléments en positions impairs et pairs, respectivement.

```
>>> :l "Elements"
[1 of 1] Compiling Main ( Elements.hs, interpreted )
Ok, modules loaded: Main.
>>> :type oddElements
oddElements :: [t] -> [t]
>>> :type evenElements
evenElements :: [t] -> [t]
>>> oddElements []
[]
```

```
>>> oddElements [1]
[1]
>>> oddElements [1..10]
[1,3,5,7,9]
>>> evenElements []
[]
>>> evenElements [1]
[]
>>> evenElements [1..10]
[2,4,6,8,10]
>>>
```

**Solution:**

```
oddElements :: [t] -> [t]
oddElements [] = []
oddElements (x:xs) = x:(evenElements xs)

evenElements :: [t] -> [t]
evenElements [] = []
evenElements (_:xs) = oddElements xs
```

- (g) Considérons le programme `Take.hs` qui propose deux implémentations de la fonction `take` :

```
take' 0 _ = []
take' _ [] = []
take' n (x:xs) = x : take' (n-1) xs

take'' _ [] = []
take'' 0 _ = []
take'' n (x:xs) = x : take'' (n-1) xs
```

Discuter des deux implémentations.

**Question 8: ghci**

Considérons le programme `OddEven.hs` suivant :

```
even' 0 = True
even' n = odd' (n - 1)

odd' 0 = False
odd' n = even' (n - 1)
```

Ouvrons une session `ghci` :

```
>>> :l "OddEven"
[1 of 1] Compiling Main ( OddEven.hs, interpreted )
Ok, modules loaded: Main.
>>> :break even'
Breakpoint 0 activated at OddEven.hs:(1,1)-(2,21)
>>> :break odd'
Breakpoint 1 activated at OddEven.hs:(4,1)-(5,20)
```

```
>>> :set stop :list
>>> even' 4
Stopped at OddEven.hs:(1,1)-(2,21)
_result :: Bool = _
1  even' 0 = True
2  even' n = odd' (n -1)
3
[OddEven.hs:(1,1)-(2,21)] >>> :step
Stopped at OddEven.hs:2:11-21
_result :: Bool = _
n :: Integer = 4
1  even' 0 = True
2  even' n = odd' (n -1)
3
[OddEven.hs:2:11-21] >>> :step
Stopped at OddEven.hs:(4,1)-(5,20)
_result :: Bool = _
3
4  odd' 0 = False
5  odd' n = even' (n-1)
[OddEven.hs:(4,1)-(5,20)] >>>
```

Que se passe-t-il? La lecture de [https://downloads.haskell.org/~ghc/7.8.3/docs/html/users\\_guide/ghci-debugger.html](https://downloads.haskell.org/~ghc/7.8.3/docs/html/users_guide/ghci-debugger.html) est -plus- que conseillée.