Haskell (IR3) – Listes

Stéphane Vialette

23 décembre 2014

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 `++`.

(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]`

Solution:

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

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

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

Solution:

```
Prelude> last [1..5]
5
Prelude> let last' xs = head (reverse xs)
Prelude> last' [1..5]
5
Prelude> let last'' xs = xs !! (length xs - 1)
Prelude> last'' [1..5]
5
Prelude> let last''' xs = head (drop (length xs - 1) xs)
Prelude> last''' [1..5]
5
Prelude>
```

(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 `!!`.

Solution:

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

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:

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

(b) Comment tester si une chaîne de caractères est un palindrome ? (Accents et majuscules ne sont pas utilisés ici.)

Solution:

```
Prelude> let removeBlanks xs = [x | x <- xs, x /= ' ']
Prelude> removeBlanks "   " == reverse (removeBlanks "   ")
True
Prelude> removeBlanks "r ad a r" == reverse (removeBlanks "r ad a r")
True
Prelude> removeBlanks "t ot o" == reverse (removeBlanks "t ot o")
False
Prelude>
```

(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:

```
Prelude> let palindrome xs = xs == reverse xs
Prelude> :t palindrome
palindrome :: Eq a => [a] -> Bool
Prelude> palindrome ""
True
Prelude> palindrome "radar"
True
Prelude> palindrome "toto"
False
Prelude>
```

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

Solution:

```
Prelude> let palindrome' xs = removeBlanks xs == reverse (removeBlanks xs)
Prelude> :t palindrome'
palindrome' :: [Char] -> Bool
Prelude> palindrome' ""
True
Prelude> palindrome' "r ad a r"
True
Prelude> palindrome' "t ot o"
False
Prelude>
```

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:

```
Prelude> :t ['a','b','c']
['a','b','c'] :: [Char]
Prelude> :t [1, 2, 3]
[1, 2, 3] :: Num t => [t]
Prelude> :t [['a','b'],['c','d']]
[['a','b'],['c','d']] :: [[Char]]
Prelude> :t [['1','2'],['3','4']]
[['1','2'],['3','4']] :: [[Char]]
Prelude> :t ('a','b')
('a','b') :: (Char, Char)
Prelude> :t ('a','b','c')
('a','b','c') :: (Char, Char, Char)
Prelude> :t (1,2)
(1,2) :: (Num t1, Num t) => (t, t1)
Prelude> :t (1,2,3)
(1,2,3) :: (Num t2, Num t1, Num t) => (t, t1, t2)
Prelude> :t [(False ,'0'),(True ,'1')]
[(False ,'0'),(True ,'1')] :: [(Bool, Char)]
Prelude> :t ([False,True],['0','1'])
([False,True],['0','1']) :: ([Bool], [Char])
Prelude> :t [tail,init,reverse]
[tail,init,reverse] :: [[a] -> [a]]
Prelude> :t ([tail,init,reverse],[take,drop])
([tail,init,reverse],[take,drop])
  :: ([[a] -> [a]], [Int -> [a1] -> [a1]])
Prelude>
```

(b) Expliquer la session suivante :

```
Prelude Data.List> :t (head, take)
(head, take) :: ([a] -> a, Int -> [a1] -> [a1])
Prelude Data.List> :t [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 -> [a1] -> [a1]`. Le type de `(head, take)` est donc `([a] -> a, Int -> [a1] -> [a1])`. Aucune difficulté ici.
Par contre une liste une liste ne peut contenir que des éléments de même type, ce qui, à l'évidence, n'est pas le cas dans `[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:

```
Prelude> let second xs = head (tail xs)
Prelude> :t second
second :: [a] -> a
Prelude> let appl (f,x) = f x
Prelude> :t appl
appl :: (t1 -> t, t1) -> t
Prelude> let pair x y = (x,y)
Prelude> :t pair
pair :: t -> t1 -> (t, t1)
Prelude> let mult x y = x * y
Prelude> :t mult
mult :: Num a => a -> a -> a
Prelude> let double = mult 2
Prelude> :t double
double :: Num a => a -> a
Prelude> let palindrome xs = reverse xs == xs
Prelude> :t palindrome
palindrome :: Eq a => [a] -> Bool
Prelude> let twice f x = f (f x)
Prelude> :t twice
twice :: (t -> t) -> t -> t
Prelude> let incrAll xs = map (+1) xs
Prelude> :t incrAll
incrAll :: Num b => [b] -> [b]
Prelude> let norme xs = sqrt (sum (map f xs)) where f x = x^2
Prelude> :t norme
norme :: Floating a => [a] -> a
Prelude>
```

(b) Quelles sont les fonctions à deux arguments ? Si une telle fonction n'est pas en forme currifiée, donnez une définition équivalente qui soit en forme currifiée.

Solution:
Prelude¿ let appl' f x = f x Prelude¿

(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.

```
Prelude> let increment x = x + 1
Prelude> :t increment
increment :: Num a => a -> a
Prelude> appl (increment, 1)
2
Prelude> appl' increment 1
2
Prelude> twice increment 1
3
Prelude>
```

(d) Quelles sont les *fonctions polymorphes* ?

Question 6: Compréhensions de listes

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

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

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

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

(c) À l'aide d'une compréhension de liste, calculer la liste des paires d'entiers (n, m), $1 \leq n \leq m \leq 100$ et $\sum_i^n i = m$.

Solution:

```
Prelude> [(m,n) | n <- [1..10],
                  m <- [1..100],
                  n <= m,
                  sum [i^2 | i <-[1..n]] == m]
[(1,1),(5,2),(14,3),(30,4),(55,5),(91,6)]
Prelude> [(m,n) | n <- [1..10],
                  m <- [n..100],
                  sum [i^2 | i <-[1..n]] == m]
[(1,1),(5,2),(14,3),(30,4),(55,5),(91,6)]
Prelude>
```

(d) 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:

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

(e) En arithmétique, deux nombres entiers distincts n et m sont dits *amicaux* (ou *aimables* ou *amiables*) 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 pairs (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 :

```
Prelude> :{
Prelude| let length' [] = 0
Prelude|     length' (_:xs) = 1 + length' xs
Prelude| :}
Prelude> :t length'
length' :: Num a => [t] -> a
Prelude> length' []
0
Prelude> length' ['a']
1
Prelude> length' ['a'..'z']
26
Prelude>
```

Une version avec accumulateur et fonction auxiliaire :

```
length'' = lengthAux 0
  where
    lengthAux n []     = n
    lengthAux n (_:xs) = lengthAux (n+1) xs
```

(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 (grace, bien sûr, à l'utilisation de l'opérateur `:`)

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

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

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

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

Avec une fonction récursive :

```
countVowels' [] = 0
countVowels' (x:xs) | vowel x    = 1 + countVowels' xs
                    | otherwise  = countVowels' xs
  where
    vowel 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.

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

Proposez une implémentation de `splitAt`.

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

```
splitAt' n xs = (take n xs, drop n xs)
```

Une seconde solution récusive :

```
splitAt'' 0 xs      = ([], xs)
splitAt'' n xs = splitAtAux n ([], xs)
  where
    splitAtAux 0 (xs, xs') = (xs, 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 :

```
fibonacci' n = fibonacciAux !! n
  where
    fibonacciAux = 0 : 1 : next fibonacciAux
    next (a : t@(b:_)) = (a+b) : next t
```

ou encore de façon élégante :

```
fibonacci'' n = fibonacciAux !! n
  where
    fibonacciAux = 0 : 1 : zipWith (+) fibonacciAux (tail fibonacciAux)
```

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

```
Prelude> :l "Elements"
[1 of 1] Compiling Main  ( Elements.hs, interpreted )
Ok, modules loaded: Main.
*Main> :t oddElements
oddElements :: [t] -> [t]
*Main> :t evenElements
evenElements :: [t] -> [t]
*Main> oddElements []
[]
*Main> oddElements [1]
[1]
*Main> oddElements [1..10]
[1,3,5,7,9]
*Main> evenElements []
[]
*Main> evenElements [1]
[]
*Main> evenElements [1..10]
[2,4,6,8,10]
*Main>
```

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` :

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

Que se passe-t-il ? La lecture de `https://downloads.haskell.org/~ghc/7.8.3/docs/html/users_guide/ghci-debugger.html` est -plus- que conseillée.