Haskell (IR3) – Listes

Stéphane Vialette (stephane.vialette@univ-eiffel.fr)

8 novembre 2022

Question 1: Expressions de liste

Toutes les fonctions du module `Data.List` ne sont pas chargées par défaut dans `Prelude`. Utilisez `import Data.List` (ou `import qualified Data.List` pour charger le module qualifié, ou encore `import qualified Data.List as L` pour charger le module qualifié par `L`) pour charger toutes les fonctions du module.

(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]`
- `1 : reverse [2..5]`

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`.

(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`.

(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

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.)

(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.)

(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 ?

(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 ?

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])`.

(b) Expliquer la session suivante :

```
>>> import Data.List
>>> :type (head, take)
(head, take) :: ([a] -> a, Int -> [a1] -> [a1])
>>> :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>
```

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.

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

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

(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.

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

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

(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 $\sum_{i=1}^{n} i = m$.

(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 ?

(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 pairs (n, m), $1\,000 \leq n < m < 1\,300$, de nombres amicaux.

Question 7: Fonctions simples

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

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

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

(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`.

(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.

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

```
>>> :load "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]
>>>
```

(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` est -plus- que conseillée.