Exercitium (curso 2013–14)

Ejercicios de programación funcional con Haskell

José A. Alonso Jiménez

Grupo de Lógica Computacional
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

Sevilla, 11 de diciembre de 2018

Para Guiomar

Índice general

Introducción

> *"The chief goal of my work as an educator and author is to help people learn to write beautiful programs."*
>
> (Donald Knuth en Computer programming as an art)

Este libro es una recopilación de las soluciones de los ejercicios propuestos en el blog Exercitium [1] durante el curso 2013–14.

El principal objetivo de Exercitium es servir de complemento a la asignatura de Informática [2] de 1º del Grado en Matemáticas de la Universidad de Sevilla.

Con los problemas de Exercitium, a diferencias de los de las relaciones [3], se pretende practicar con los conocimientos adquiridos durante todo el curso, mientras que con las relaciones están orientadas a los nuevos conocimientos.

Habitualmente de cada ejercicio se muestra distintas soluciones y se compara sus eficiencias.

La dinámica del blog es la siguiente: cada día, de lunes a viernes, se propone un ejercicio para que los alumnos escriban distintas soluciones en los comentarios. Pasado 7 días de la propuesta de cada ejercicio, se cierra los comentarios y se publica una selección de sus soluciones.

Para conocer la cronología de los temas explicados se puede consultar el diario de clase [4].

El código del libro se encuentra en GitHub [5]

[1] https://www.glc.us.es/~jalonso/exercitium
[2] https://www.cs.us.es/~jalonso/cursos/i1m-13
[3] https://www.cs.us.es/~jalonso/cursos/i1m-13/ejercicios/ejercicios-I1M-2013.pdf
[4] https://www.glc.us.es/~jalonso/vestigium/category/curso/i1m/i1m2013
[5] https://github.com/jaalonso/Exercitium2013

Ejercicio 1

Iguales al siguiente

Ejercicio propuesto el 21 de Abril de 2014

Definir la función

```
igualesAlSiguiente :: Eq a => [a] -> [a]
```

tal que (igualesAlSiguiente xs) es la lista de los elementos de xs que son iguales a su siguiente. Por ejemplo,

```
igualesAlSiguiente [1,2,2,2,3,3,4]  ==  [2,2,3]
igualesAlSiguiente [1..10]          ==  []
```

Soluciones

```
import Data.List (group)
import Test.QuickCheck

-- 1ª definición (por comprensión):
igualesAlSiguiente :: Eq a => [a] -> [a]
igualesAlSiguiente xs =
  [x | (x,y) <- zip xs (tail xs), x == y]

-- 2ª definición (por recursión):
igualesAlSiguiente2 :: Eq a => [a] -> [a]
```

```haskell
igualesAlSiguiente2 (x:y:zs)
  | x == y    = x : igualesAlSiguiente2 (y:zs)
  | otherwise = igualesAlSiguiente2 (y:zs)
igualesAlSiguiente2 _ = []

-- 3ª definición (con concat y comprensión):
igualesAlSiguiente3 :: Eq a => [a] -> [a]
igualesAlSiguiente3 xs = concat [ys | (_:ys) <- group xs]

-- 4ª definición (con concat y map):
igualesAlSiguiente4 :: Eq a => [a] -> [a]
igualesAlSiguiente4 xs = concat (map tail (group xs))

-- 5ª definición (con concatMap):
igualesAlSiguiente5 :: Eq a => [a] -> [a]
igualesAlSiguiente5 xs = concatMap tail (group xs)

-- 6ª definición (con concatMap y sin argumentos):
igualesAlSiguiente6 :: Eq a => [a] -> [a]
igualesAlSiguiente6 = concatMap tail . group

-- Equivalencia
-- ============

-- La propiedad es
prop_igualesAlSiguiente_equiv :: [Int] -> Bool
prop_igualesAlSiguiente_equiv xs =
  igualesAlSiguiente xs == igualesAlSiguiente2 xs &&
  igualesAlSiguiente xs == igualesAlSiguiente3 xs &&
  igualesAlSiguiente xs == igualesAlSiguiente4 xs &&
  igualesAlSiguiente xs == igualesAlSiguiente5 xs &&
  igualesAlSiguiente xs == igualesAlSiguiente6 xs

verifica_igualesAlSiguiente_equiv :: IO ()
verifica_igualesAlSiguiente_equiv =
  quickCheck prop_igualesAlSiguiente_equiv

-- La comprobación es
--    λ> verifica_igualesAlSiguiente_equiv
--    +++ OK, passed 100 tests.
```

```
--    (0.07 secs, 9,911,528 bytes)
```

Ejercicio 2

Ordenación por el máximo

Ejercicio propuesto el 22 de Abril de 2014

Definir la función

```
ordenadosPorMaximo :: Ord a => [[a]] -> [[a]]
```

tal que (ordenadosPorMaximo xss) es la lista de los elementos de xss ordenada por sus máximos. Por ejemplo,

```
ghci> ordenadosPorMaximo [[3,2],[6,7,5],[1,4]]
[[3,2],[1,4],[6,7,5]]
ghci> ordenadosPorMaximo ["este","es","el","primero"]
["el","primero","es","este"]
```

Soluciones

```
import Data.List (sort)
import GHC.Exts  (sortWith)
import Test.QuickCheck

-- 1ª definición
ordenadosPorMaximo :: Ord a => [[a]] -> [[a]]
ordenadosPorMaximo xss =
  map snd (sort [(maximum xs,xs) | xs <- xss])
```

```haskell
-- 2ª definición
ordenadosPorMaximo2 :: Ord a => [[a]] -> [[a]]
ordenadosPorMaximo2 xss =
  [xs | (_,xs) <- sort [(maximum xs,xs) | xs <- xss]]

-- 3ª definición:
ordenadosPorMaximo3 :: Ord a => [[a]] -> [[a]]
ordenadosPorMaximo3 = sortWith maximum

-- Equivalencia
-- ============

verificaOrdenadosPorMaximo :: IO ()
verificaOrdenadosPorMaximo =
  quickCheck prop_ordenadosPorMaximo

prop_ordenadosPorMaximo :: [[Int]] -> Bool
prop_ordenadosPorMaximo xs =
  ordenadosPorMaximo ys == ordenadosPorMaximo2 ys
  where ys = filter (not . null) xs

-- Comprobación
--    λ> verificaOrdenadosPorMaximo
--    +++ OK, passed 100 tests.
```

Ejercicio 3

La bandera tricolor

Ejercicio propuesto el 23 de Abril de 2014

El problema de la bandera tricolor consiste en lo siguiente: Dada un lista de objetos xs que pueden ser rojos, amarillos o morados, se pide devolver una lista ys que contiene los elementos de xs, primero los rojos, luego los amarillos y por último los morados.

Se pide definir el tipo de dato Color para representar los colores con los constructores R, A y M correspondientes al rojo, amarillo y morado y la función

```
banderaTricolor :: [Color] -> [Color]
```

tal que (banderaTricolor xs) es la bandera tricolor formada con los elementos de xs. Por ejemplo,

```
bandera [M,R,A,A,R,R,A,M,M]  ==  [R,R,R,A,A,A,M,M,M]
bandera [M,R,A,R,R,A]        ==  [R,R,R,A,A,M]
```

Soluciones

```
import Data.List (sort)
import Test.QuickCheck

data Color = R | A | M
```

```haskell
  deriving (Show, Eq, Ord, Enum)

-- 1ª definición (con sort):
banderaTricolor :: [Color] -> [Color]
banderaTricolor = sort

-- 2ª definición (por comprensión):
banderaTricolor2 :: [Color] -> [Color]
banderaTricolor2 xs =
  [x | x <- xs, x == R] ++ [x | x <- xs, x == A] ++ [x | x <- xs, x == M]

-- 3ª definición (por comprensión y concat):
banderaTricolor3 :: [Color] -> [Color]
banderaTricolor3 xs =
  concat [[x | x <- xs, x == c] | c <- [R,A,M]]

-- 4ª definición (por recursión):
banderaTricolor4 :: [Color] -> [Color]
banderaTricolor4 xs = aux xs ([],[],[])
  where aux []     (rs,as,ms) = rs ++ as ++ ms
        aux (R:ys) (rs,as,ms) = aux ys (R:rs,   as,   ms)
        aux (A:ys) (rs,as,ms) = aux ys (  rs, A:as,   ms)
        aux (M:ys) (rs,as,ms) = aux ys (  rs,   as, M:ms)

-- 5ª definición (por recursión):
banderaTricolor5 :: [Color] -> [Color]
banderaTricolor5 xs = aux xs (0,0,0)
  where aux []     (as,rs,ms) = replicate rs R ++
                                replicate as A ++
                                replicate ms M
        aux (A:ys) (as,rs,ms) = aux ys (1+as,   rs,   ms)
        aux (R:ys) (as,rs,ms) = aux ys (  as, 1+rs,   ms)
        aux (M:ys) (as,rs,ms) = aux ys (  as,   rs, 1+ms)

-- Equivalencia
-- ============

instance Arbitrary Color where
  arbitrary = elements [R,A,M]
```

```haskell
prop_banderaTricolor :: [Color] -> Bool
prop_banderaTricolor xs =
  all (== banderaTricolor xs)
      [f xs | f <- [ banderaTricolor2
                   , banderaTricolor3
                   , banderaTricolor4
                   , banderaTricolor5]]

verifica_banderaTricolor :: IO ()
verifica_banderaTricolor =
  quickCheck prop_banderaTricolor

-- La comprobación es
--    λ> verifica_banderaTricolor
--    +++ OK, passed 100 tests.

-- Comparaciones:
--    ghci> bandera n = concat [replicate n c | c <- [M,R,A]]
--
--    ghci> length (banderaTricolor (bandera 1000000))
--    3000000
--    (2.65 secs, 312128100 bytes)
--
--    ghci> length (banderaTricolor2 (bandera 1000000))
--    3000000
--    (7.78 secs, 512387912 bytes)
--
--    ghci> length (banderaTricolor3 (bandera 1000000))
--    3000000
--    (7.84 secs, 576080444 bytes)
--
--    ghci> length (banderaTricolor4 (bandera 1000000))
--    3000000
--    (3.76 secs, 476484220 bytes)
--
--    ghci> length (banderaTricolor5 (bandera 1000000))
--    3000000
--    (4.45 secs, 622205356 bytes)
```

Ejercicio 4

Elementos minimales

Ejercicio propuesto el 24 de Abril de 2014

Definir la función

```
minimales :: Eq a => [[a]] -> [[a]]
```

tal que (minimales xss) es la lista de los elementos de xss que no están contenidos en otros elementos de xss. Por ejemplo,

```
minimales [[1,3],[2,3,1],[3,2,5]]       ==  [[2,3,1],[3,2,5]]
minimales [[1,3],[2,3,1],[3,2,5],[3,1]]  ==  [[2,3,1],[3,2,5]]
```

Soluciones

```haskell
import Data.List (delete, nub)

minimales :: Eq a => [[a]] -> [[a]]
minimales xss =
  [xs | xs <- xss, [ys | ys <- xss, subconjuntoPropio xs ys] == []]

-- (subconjuntoPropio xs ys) se verifica si xs es un subconjunto propio
-- de ys. Por ejemplo,
--    subconjuntoPropio [1,3] [3,1,3]    ==  False
--    subconjuntoPropio [1,3,1] [3,1,2]  ==  True
```

```haskell
subconjuntoPropio :: Eq a => [a] -> [a] -> Bool
subconjuntoPropio xs ys = subconjuntoPropio' (nub xs) (nub ys)
  where
    subconjuntoPropio' _  [] = False
    subconjuntoPropio' [] _  = True
    subconjuntoPropio' (x:xs') ys' =
      x `elem` ys' && subconjuntoPropio xs' (delete x ys')
```

Ejercicio 5

Mastermind

Ejercicio propuesto el 25 de Abril de 2014

El Mastermind es un juego que consiste en deducir un código numérico formado por una lista de números distintos. Cada vez que se empieza una partida, el programa debe elegir un código, que será lo que el jugador debe adivinar en la menor cantidad de intentos posibles. Cada intento consiste en una propuesta de un código posible que propone el jugador, y una respuesta del programa. Las respuestas le darán pistas al jugador para que pueda deducir el código.

Estas pistas indican cuán cerca estuvo el número propuesto de la solución a través de dos valores: la cantidad de aciertos es la cantidad de dígitos que propuso el jugador que también están en el código en la misma posición. La cantidad de coincidencias es la cantidad de digitos que propuso el jugador que también están en el código pero en una posición distinta.

Por ejemplo, si el código que eligió el programa es el [2,6,0,7], y el jugador propone el [1,4,0,6], el programa le debe responder un acierto (el 0, que está en el código original en el mismo lugar, el tercero), y una coincidencia (el 6, que también está en el código original, pero en la segunda posición, no en el cuarto como fue propuesto). Si el jugador hubiera propuesto el [3,5,9,1], habría obtenido como respuesta ningún acierto y ninguna coincidencia, ya que no hay números en común con el código original, y si se obtienen cuatro aciertos es porque el jugador adivinó el código y ganó el juego.

Definir la función

```
mastermind :: [Int] -> [Int] -> (Int,Int)
```

tal que (mastermind xs ys) es el par formado por los números de aciertos y de coincidencias entre xs e ys. Por ejemplo,

```
mastermind [2,6,0,7] [1,4,0,6]  ==  (1,1)
mastermind [2,6,0,7] [3,5,9,1]  ==  (0,0)
mastermind [2,6,0,7] [1,6,0,4]  ==  (2,0)
mastermind [2,6,0,7] [2,6,0,7]  ==  (4,0)
```

Soluciones

```
import Test.QuickCheck
import Data.List (nub)

-- 1ª solución (por comprensión):
mastermind :: [Int] -> [Int] -> (Int,Int)
mastermind xs ys =
  (length (aciertos xs ys),length (coincidencias xs ys))

-- (aciertos xs ys) es la lista de aciertos entre xs e ys. Por ejemplo,
--    aciertos [2,6,0,7] [1,4,0,6]  ==  [0]
aciertos :: Eq a => [a] -> [a] -> [a]
aciertos xs ys = [x | (x,y) <- zip xs ys, x == y]

-- (coincidencia xs ys) es la lista de coincidencias entre xs e ys. Por
-- ejemplo,
--    coincidencias [2,6,0,7] [1,4,0,6]  ==  [6]
coincidencias :: Eq a => [a] -> [a] -> [a]
coincidencias xs ys =
  [x | x <- xs, x `elem` ys, x `notElem` zs]
  where zs = aciertos xs ys

-- 2ª solución (por recursión):
mastermind2 :: [Int] -> [Int] -> (Int,Int)
mastermind2 xs ys = aux xs ys
  where aux [] [] = (0,0)
        aux (x:xs') (z:zs)
            | x == z      = (a+1,b)
            | x `elem` ys = (a,b+1)
            | otherwise   = (a,b)
```

```
           where (a,b) = aux xs' zs
        aux _ _ = error "Imposible"

-- 3ª solución:
mastermind3 :: [Int] -> [Int] -> (Int,Int)
mastermind3 xs ys = (nAciertos,nCoincidencias)
    where nAciertos = length [(x,y) | (x,y) <- zip xs ys, x == y]
          nCoincidencias = length (xs++ys) - length (nub (xs++ys)) - nAciertos

-- Equivalencia
-- ============

prop_mastermind :: [Int] -> [Int] -> Bool
prop_mastermind xs ys =
  all (== mastermind cs ds)
      [f cs ds | f <- [ mastermind2
                      , mastermind3]]
  where as = nub xs
        bs = nub ys
        n  = min (length as) (length bs)
        cs = take n as
        ds = take n bs

verifica_mastermind :: IO ()
verifica_mastermind =
  quickCheck prop_mastermind

-- La comprobación es
--    λ> verifica_mastermind
--    +++ OK, passed 100 tests.
```

Ejercicio 6

Primos consecutivos con media capicúa

Ejercicio propuesto el 28 de Abril de 2014

La pasada semana, Antonio Roldán publicó en Twitter la siguiente observación:

> Los pares de primos consecutivos (97,101) y (109,113) son los más pequeños, con promedio capicúa con más de una cifra: (97+101)/2=99 y (109+113)/2=111.

A partir de ella, se propone el ejercicio de hoy.

Definir la constante

```
primosConsecutivosConMediaCapicua :: [(Int,Int,Int)]
```

tal que primosConsecutivosConMediaCapicua es la lista de las ternas (x,y,z) tales que x e y son primos consecutivos tales que su media, z, es capicúa. Por ejemplo,

```
ghci> take 5 primosConsecutivosConMediaCapicua
[(3,5,4),(5,7,6),(7,11,9),(97,101,99),(109,113,111)]
```

Calcular cuántos hay anteriores a 2014.

Soluciones

```haskell
import Data.Numbers.Primes (primes)

primosConsecutivosConMediaCapicua :: [(Int,Int,Int)]
primosConsecutivosConMediaCapicua =
  [(x,y,z) | (x,y) <- zip (tail primes) (tail (tail primes))
           , let z = (x + y) `div` 2
           , capicua z]

-- (capicua x) se verifica si x es capicúa. Por ejemplo,
capicua :: Int -> Bool
capicua x = ys == reverse ys
  where ys = show x

-- El cálculo es
-- λ> length (takeWhile (\(_,y,_) -> y < 2014) primosConsecutivosConMediaCapicua)
--  20
```

Ejercicio 7

Anagramas

Ejercicio propuesto el 29 de Abril de 2014

Una palabra es una anagrama de otra si se puede obtener permutando sus letras. Por ejemplo, mora y roma son anagramas de amor.

Definir la función

```
anagramas :: String -> [String] -> [String]
```

tal que (anagramas x ys) es la lista de los elementos de ys que son anagramas de x. Por ejemplo,

```
ghci> anagramas "amor" ["Roma","mola","loma","moRa", "rama"]
["Roma","moRa"]
ghci> anagramas "rama" ["aMar","amaRa","roMa","marr","aRma"]
["aMar","aRma"]
```

Soluciones

```
import Data.List     (sort)
import Data.Char     (toLower)
import Data.Function (on)

-- 1ª definición (por recursión)
```

```haskell
-- ============================

anagramas :: String -> [String] -> [String]
anagramas _ [] = []
anagramas x (y:ys)
  | sonAnagramas x y = y : anagramas x ys
  | otherwise        = anagramas x ys

-- (sonAnagramas xs ys) se verifica si xs e ys son anagramas. Por
-- ejemplo,
--    sonAnagramas "amor" "Roma"  ==  True
--    sonAnagramas "amor" "mola"  ==  False
sonAnagramas :: String -> String -> Bool
sonAnagramas xs ys =
  sort (map toLower xs) == sort (map toLower ys)

-- 2ª definición de sonAnagramas
sonAnagramas2 :: String -> String -> Bool
sonAnagramas2 xs ys =
  (sort . map toLower) xs == (sort . map toLower) ys

-- 3ª definición de sonAnagramas (con on)
sonAnagramas3 :: String -> String -> Bool
sonAnagramas3 = (==) `on` (sort . map toLower)

-- Nota. En la definición anterior se usa la función on ya que
--    (f `on` g) x y
-- es equivalente a
--    f (g x) (g y)
-- Por ejemplo,
--    ghci> ((*) `on` (+2)) 3 4
--    30

-- 2ª definición (por comprensión)
-- ==============================

anagramas2 :: String -> [String] -> [String]
anagramas2 x ys = [y | y <- ys, sonAnagramas x y]

-- 3ª definición (con filter y sin el 2ª argumento)
```

```haskell
-- ================================================

anagramas3 :: String -> [String] -> [String]
anagramas3 x = filter (`sonAnagramas` x)

-- 4ª definición (sin sonAnagramas ni el 2º argumento)
-- ================================================

anagramas4 :: String -> [String] -> [String]
anagramas4 x = filter (((==) `on` (sort . map toLower)) x)
```