Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI3661 - Taller de Lenguajes de Programación I
Enero-Marzo 2016

Quiz Haskell

(15 puntos)

Nombre:

Carnet:

Atención: en todas sus respuestas debe utilizar *exclusivamente* funciones disponibles en el Preludio Haskell. Si Ud. no sabe si una función está o no en el Preludio Haskell, puede preguntarlo *públicamente* para recibir un **si** o **no** como respuesta y **nada** más.

1. **(3 puntos)** Para cada una de las siguientes expresiones calcule el tipo más general posible:

   a) `map not . map odd`

   ```
   Integral a => [a] -> [Bool]
   ```

   No tiene sentido aplicar la función `odd` sobre cualquier número, por tanto **no es** `Num a`.

   b) `\f p -> f (fst p) (snd p)`

   ```
   (a -> b -> c) -> (a, b) -> c
   ```

   No es más que la función estándar `uncurry` escrita como $\lambda-$abstracción.

   c) `wat f q x = q : (if null x then [] else wat f (f q (head x)) (tail x))`

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

   No es otra sino la función estándar `scanl`.

2. **(6 puntos)** Para la tarea de programación, Ud. tuvo que haber implantado la función

   ```
   shuffle :: StdGen -> Hand -> Hand
   ```

   para barajar el mazo. En el enunciado se proponía un algoritmo particular para barajado. En la clase del 2016-02-12 describí otro algoritmo diferente, inspirado en *Merge Sort*. Para la tarea de programación, podía escogerse uno cualquiera de los dos algoritmos. Implante aquí la función `shuffle`, pero con el algoritmo que Ud. **NO** escogió para la tarea.

   - Según el algoritmo inspirado en *Merge Sort*.

```
halve :: [a] -> ([a],[a])
halve []     = ([],[])
halve (x:xs) = (x:h1,h0)
  where (h0,h1) = halve xs
shuffle :: StdGen -> Hand -> Hand
shuffle gen (H cards) = H $ doit gen cards
  where
    doit _ []  = []
    doit _ [c] = [c]
    doit g cs  = H $ merge (doit g0 h0) (doit g1 h1)
      where
        (h0,h1)             = halve cs
        (g0,g1)             = split g
        merge []     ys     = ys
        merge xs     []     = xs
        merge (x:xs) (y:ys) = if yes then x : merge xs (y:ys)
                                     else y : merge (x:xs) ys
                                       where (yes,_) = random g
```

- Según el algoritmo sugerido en el enunciado – un `foldl` con un uso legítimo de `length`

```
shuffle :: StdGen -> Hand -> Hand
shuffle g (H cs) = H $ fst $ foldl' pick ([],(g,cs,length cs)) cs
  where pick (sh,(g0,b,n)) _ = ((b !! p):sh,(g1,b',n-1))
        where (p,g1) = randomR (0,n-1) g0
              b'     = take p b ++ drop (p+1) b
```

3. **(6 puntos)** (Ejercicio 26 de los "99 Haskell Problems") Considere la siguiente implantación, por demás nefasta, de una función para obtener las posibles combinaciones de n elementos tomados de una lista

```
combinations :: Int -> [a] -> [[a]]
combinations 0 _  = [[]]
combinations n xs = [ xs !! i : x | i <- [0..(length xs)-1],
                                    x <- combinations (n-1) (drop (i+1) xs) ]
```

   Proponga una mejor implantación usando funciones de orden superior.

   Hay varias maneras de resolver este problema, la mayoría utilizando `Data.List` (que no está permitido en el examen), pero hay al menos dos soluciones usando funciones de orden superior que son aceptables. Por cierto, ambas soluciones están disponibles en la sección de soluciones a los "99 Haskell Problems":

   - Empleando un `map` – es cierto que hay recursión directa y `(++)`, pero es abismalmente menos malo que usar `(!!)`, `drop` y `length`.

```
combinations 0 _      = [[]]
combinations _ []     = []
combinations n (x:xs) = (map (x:) (combinations (n-1) xs)) ++
                                  (combinations n xs)
```

   - Empleando `filter` encima de un `foldr` que genera posibles subsecuencias – de nuevo, uno de los pocos usos legítimos de `length`.

```
combinations k ns = filter ((k==).length) (subseqs ns)
  where subseqs xs             = [] : nonEmptySubseqs xs
        nonEmptySubseqs []     = []
        nonEmptySubseqs (x:xs) = [x] : foldr f [] (nonEmptySubseqs xs)
                                   where f ys r = ys : (x : ys) : r
```

4. **(3 puntos extra)** ¿Resolvió el problema de Hamming? ¡Albricias! Escriba su solución aquí.

   Esta solución está publicada en las láminas de clase de CI-3641 en relación a Recursión, que estudiamos en la clase del 2016-02-10 y entre los ejemplos de programación perezosa disponibles a propósito de la clase del 2016-02-12.

```
h = 1 : m (map (*2) h)
          (m (map (*3) h)
             (map (*5) h))
   where m (x:xs) (y:ys)
           | x == y    = x : m xs ys
           | x <  y    = x : m xs (y:ys)
           | otherwise = y : m (x:xs) ys
```