Programación Funcional - Práctico 3

Nota:

- Cuando decimos "como *foldl*" o "como *foldr*" nos referimos a que la solución debe ser construida solamente en términos del *fold* correspondiente, definiendo los argumentos que dicho *fold* requiera. Ejemplo: *sum* como *foldr*, se define como $sum = foldr\ (+)\ 0$
- Cuando decimos "usando *foldl*" o "usando *foldr*" nos referimos a que la solución contiene un *fold* como parte de ella. Ejemplo: *sumaCero* usando *foldr*, se define como $sumaCero = (0 ==)\ .\ foldr\ (+)\ 0$.

1. Explique el tipo de las siguientes funciones:
   (a) $min\ x\ y = \textbf{if}\ x < y\ \textbf{then}\ x\ \textbf{else}\ y$
   (b) $paren\ x = \texttt{"("} +\!\!+ show\ x +\!\!+ \texttt{")"}$
2. Dada la siguiente definición de tipo:

   $$\textbf{data}\ Semaforo = Verde \mid Amarillo \mid Rojo$$

   ¿Qué falta para que sea posible hacer $(show\ Verde)$?
3. Defina las siguientes funciones usando recursión explícita.
   (a) $sumSqs :: Num\ a \Rightarrow [a] \to a$
   Suma los cuadrados de los elementos de una lista.
   (b) $elem :: Eq\ a \Rightarrow a \to [a] \to Bool$
   Determina si un elemento pertenece a una lista.
   (c) $elimDups :: Eq\ a \Rightarrow [a] \to [a]$
   Elimina los duplicados adyacentes de una lista.
   Por ejemplo, $elimDups\ [1,2,2,3,4,4,4,3]$ retorna $[1,2,3,4,3]$.
   (d) $split :: [a] \to ([a],[a])$
   Divide una lista en dos listas colocando sus elementos de forma alternada. Por ejemplo, $split\ [2,4,6,8,7]$ retorna $([2,6,7],[4,8])$.
   (e) $maxInd :: Ord\ a \Rightarrow [a] \to (a, Int)$
   Retorna el máximo de una lista no vacía y el índice de su primera ocurrencia. Los índices se comienzan a numerar en 0. Por ejemplo, $maxInd\ [8,10,6,10,10]$ retorna $(10,1)$.

(f) $merge :: Ord\ a \Rightarrow [a] \to [a] \to [a]$
Mezcla dos listas ordenadas en una nueva lista ordenada.

4. Defina las siguientes funciones como *foldr*:
   (a) $sumSqs :: Num\ a \Rightarrow [a] \to a$
   (b) $elimDups :: Eq\ a \Rightarrow [a] \to [a]$
   (c) $split :: [a] \to ([a], [a])$
   (d) $takeWhile :: (a \to Bool) \to [a] \to [a]$

5. Defina las siguientes funciones por recursión estructural usando tail-recursion.
   (a) $sumSqs :: Num\ a \Rightarrow [a] \to a$
   (b) $elem :: Eq\ a \Rightarrow a \to [a] \to Bool$
   (c) $elimDups :: Eq\ a \Rightarrow [a] \to [a]$
   (d) $split :: [a] \to ([a], [a])$
   (e) $maxInd :: Ord\ a \Rightarrow [a] \to (a, Int)$
   (f) $takeWhile :: (a \to Bool) \to [a] \to [a]$
   (g) $dropWhile :: (a \to Bool) \to [a] \to [a]$

6. Defina la función *sumSqs* como *foldl*.

7. Sea $h\ x\ xs = x - sum\ xs$. ¿Cuál de las siguientes afirmaciones es correcta?
   (a) $h\ x\ xs = foldr\ (-)\ x\ xs$
   (b) $h\ x\ xs = foldl\ (-)\ x\ xs$

8. Una buena implementación de *elem como foldr* es más eficiente que una implementación de *elem como foldl*. Defina *elem* como *foldr* y *como foldl*. Compare las implementaciones ejecutando con los argumentos 1 y $[1\,..\,10000000]$. Explique por qué una es más eficiente que la otra.

9. Defina la función *maxInd usando foldr* y *usando foldl*.

10. Al recorrer una lista las funciones *foldr* y *foldl* pueden desarmar y reconstruír la misma. En el caso de *foldl*, la reconstrucción invierte la lista. Defina *split* y *elimDups usando foldl* y *reverse*.

11. La función *foldl* siempre recorre completamente la lista. Defina la función *takeWhile usando foldl*. Debe usar el acumulador para saber cuándo se obtuvo toda la información relevante de la lista.

12. Suponga que representamos números naturales como listas de dígitos ordenados de forma descendente según su significación. Por ejemplo [1,2,5] representa al número 125.

(a) Defina una función $sucesor :: [Int] \to [Int]$, que dado un natural en esta representación compute el siguiente natural. ¿Cuál estrategia resulta más adecuada para implementar esta función, la recursión o la recorrida con acumulador? Escriba la función como/usando *foldr* o *foldl*, siguiendo la estrategia más adecuada.

(b) Defina una función $decimal :: [Int] \to Int$, que dado un natural en esta representación compute el entero correspondiente. ¿Cuál estrategia resulta más adecuada para implementar esta función, la recursión o la recorrida con acumulador? Escriba la función como/usando *foldr* o *foldl*, siguiendo la estrategia más adecuada.

(c) Defina una función $repr :: Int \to [Int]$, que dado un natural (de tipo *Int*) retorna su representación.

13. El *algoritmo de Luhn* es una fórmula de checksum que se usa para validar datos numéricos tales como números de tarjetas de crédito o números de identificación. Dado un número natural (de tipo *Int*), tal que el dígito menos significativo corresponde al dígito de control, el algoritmo realiza los siguientes pasos para validarlo:

- Se obtiene la lista de dígitos del número ordenados de forma descendente según su significación. Por ejemplo, el número 125 resulta en $[1, 2, 5]$. Para ello podemos usar la función *repr* del ejercicio anterior.
- $doubleD :: [Int] \to [Int]$

  Se recorre la lista desde el final hacia adelante y se multiplica por dos cada segundo dígito. Por ejemplo, *dobleD* $[7, 2, 4, 5]$ resulta en $[14, 2, 8, 5]$.
- $sumaD :: [Int] \to Int$

  Se suman las posiciones de la lista; al sumarlas, las posiciones que sean mayores que 9 deben ser ajustadas restándole 9.

  Por ejemplo, *sumaD* $[14, 2, 8, 5]$ resulta en $(14 - 9) + 2 + 8 + 5$.
- $validar :: Int \to Bool$

  Si el resultado de la suma anterior es múltiplo de 10 entonces el número original es válido.

(a) Implemente las funciones *dobleD*, *sumaD* y *validar*.

(b) Defina una función

$$luhn :: Int \to Bool$$

que realice la validación de un número mediante la composición de las funciones anteriores.