Programación Funcional

Trabajo Práctico Nro. 7

Temas: Funciones de alto orden sobre listas.

Bibliografía relacionada:

- Bird, Richard. Introduction to funtional programming using Haskell. Prentice Hall, 1998 (Second Edition). Cap. 4.

1. Implementar las funciones del ejercicio 2 de la práctica 4 utilizando esquemas de recursión.

2. Definir las siguientes funciones utilizando funciones de alto orden siempre que sea posible:

`pal`,	que determina si un string es palíndromo.
`hs`,	que cuenta la cantidad de palabras que empiezan con h en una lista dada.
`avgLength`,	que calcula la longitud promedio de las palabras de una lista.
`adjacents`,	que tome una lista y retorna la lista de todos los pares ordenados de elementos adyacentes, por ejemplo: `adjacents [2, 1, 11, 4] = [(2,1),(1,11),(11,4)]`
`diffAdj`,	que toma una lista de números y devuelve la lista de los pares ordenados de todos los números adyacentes cuya diferencia es par.
`remDups`,	que devuelve una lista con los mismos elementos que la original, pero eliminando todos aquellos valores que fueran adyacentes e iguales, dejando una sola ocurrencia de cada uno.
`primes`,	que dado un entero `n` devuelve una lista con los `n` primeros primos.

3. Sea la función `f = foldr (:) []`

   - ¿Qué tipo tiene?
   - Reducir la función aplicada a una lista cualquiera.
   - Escribir una definición equivalente, pero más simple.

4. Definir la función `filter` en términos de `map` y `concat`.

5. Definir las funciones `takewhile`, que devuelve el segmento inicial más largo de una lista de elementos que verifican una condición dada, y `dropwhile`, que devuelve el segmento de la lista que comienza con el primer elemento que no verifica la condición dada.

6. ¿Recuerda los términos lambda? Escriba una función `ffreshIndex :: [Lt] -> Int` que tome una lista `ts` de términos lambda y devuelva un número `n` que para toda variable `X m` que aparece en un término `t` en `ts`, `n` > `m`.

7. Demostrar por inducción en la estructura de las listas.

   a) `map f (xs ++ ys) = map f xs ++ map f ys`

   b) `map f . concat = concat . map (map f)`

   c) `filter p (xs ++ ys) = filter p xs ++ filter p ys`

   d) `map (map f) . map (x:) = map ((f x):) . map (map f)`

   e) `concat . map concat = concat . concat`

8. ⊛ ¿Se pueden implementar `insert` y `evenPos` utilizando `foldr`?

```
insert y  []    = []
insert y (x:xs) = if x<y then x:insert y xs else y:x:xs

evenPos []       = []
evenPos [x]      = [x]
evenPos (x:y:xs) = x:evenPos xs
```

Ejercicios complementarios

9. Considerando la función:

```
inits :: [a] -> [[a]]
inits []     = []
inits (x:xs) = [x] : map (x:) (inits xs)
```

   Demostrar que `inits . map f = map (map f) . inits`

10. Demostrar por inducción en la estructura de las listas.

    a) `map (f . g) = map f . map g`

    b) `filter p . filter q = filter r where r x = p x && q x`

    c) `filter p . map f = map f . filter (p . f)`

    d) `takewhile p xs ++ dropwhile p xs = xs`