Programación Funcional

Curso 2017/2018 – Sesión práctica (Lote 2)

Esto es un repertorio de actividades sugeridas para la sesión de prácticas

1. Escribe en la consola expresiones para calcular el valor indicado en cada apartado:

   La idea aquí es utilizar en lo posible funciones primitivas, incluyendo las habituales de orden superior, o bien listas intensionales. Pero en algún caso te puede hacer falta alguna función auxiliar.

   - La lista de los cuadrados de los números naturales entre 0 y 50 (o sea, [0,1,4,9,...,2500]). Hazlo sin usar y usando listas infinitas.
   - La lista anterior, pero con cada número emparejado con su cuadrado y en orden inverso ([(50,2500),(49,2401),...,(2,4),(1,1),(0,0)])
   - La suma $\sum_{i=1}^{i=100} i \cdot \mid sen(i) \mid$
   - La lista con las 50 primeras potencias de 3. *(Como antes, hazlo sin usar y usando listas infinitas. Y, para este segundo caso, hazlo usando* `map` *y usando* `iterate`*).*
   - El número de potencias de 3 que sean menores que 10^{100} y acaben en 67.
   - La suma de los números menores que 1000 que sean múltiplos de 3 o 5.
   - La suma de los números menores que 10^{20} que sean múltiplos de 3 o 5. *(Moraleja: como siempre, más vale maña que fuerza)*
   - La lista $[[1, 2, 3, 4, \ldots, 20], [1, 4, 9, 16, \ldots, 400], [1, 8, 27, \ldots, 8000], ..., [1, 2^{10}, 3^{10}, \ldots, 20^{10}]]$
   - La lista $[[1, 1, 1, \ldots, 1], [2, 4, 8, 16, \ldots, 2^{10}], [3, 9, 27, \ldots, 3^{10}], \ldots, [20, 20^2, 20^3, \ldots, 20^{10}]]$
   - La lista $[1, -2, 3, -4, 5, -6, \ldots]$
   - La lista $[0, 1, -1, 2, -2, 3, -3, \ldots]$
   - Una lista infinita que sirva como enumeración de todas las parejas de números naturales, y una función que actúe como inversa de la enumeración, es decír, que dada una de tales parejas (n, m), calcule su posición en esa lista. Lo mismo con parejas de números enteros.

2. Programa, indicando sus tipos, las siguientes funciones (o propiedades, es decir, funciones booleanas) de orden superior, utilizando si te conviene otras primitivas o previas:

   - `filter2 xs p q` = (us, vs) donde us son los elementos de xs que cumplen p y vs los que cumplen q
   - `filters xs ps` = $[xs_1, \ldots, xs_n]$, donde xs_i son los elementos de xs que cumplen p_i, supuesto que ps es $[p_1, \ldots, p_n]$.
   - `partition p xs` = (us, vs), donde us son los elementos de xs que cumplen p y vs son el resto.
   - `span p xs` = (us, vs), donde us es el mayor prefijo de xs tal que todos sus elementos cumplen p y vs es el resto.
   - `mapx x [f0,f1,...,fn] = [f0 x,f1 x,...,fn x]`
   - `iguales f g n m` $\Leftrightarrow f\ x = g\ x$, para todo $n \leq x \leq m$
   - `cuantos p xs` = número de elementos de la lista xs que cumplen la propiedad p
   - `mayoria p xs` $\Leftrightarrow$ la mayoría de los elementos de la lista xs cumplen la propiedad p
   - `menorA n m p` = menor x con $n \leq x \leq m$ que verifica p
   - `menor n p` = menor $x \geq n$ que verifica p
   - `mayorA n m p` = mayor x con $n \leq x \leq m$ que verifica p
   - `mayor n p` = mayor $x \leq n$ que verifica p
   - `ex n m p` $\Leftrightarrow$ existe x con $n \leq x \leq m$ que verifica p
   - `pt n m p` $\Leftrightarrow$ todos los x con $n \leq x \leq m$ verifican p

3. Programa una vez más, en esta ocasión usando funciones de orden superior y/o listas intensionales las siguientes funciones o propiedades (i.e., funciones con valor `Bool` relativas a divisibilidad y primos (como siempre, si lo haces de varias formas, mejor):

   - La lista de divisores de un número.
   - La lista con los números entre 19 y 50 emparejados cada uno con la lista de sus divisores (excluido el propio número), es decir, la lista

     $$[(19,[1]),(20,[1,2,4,5,10]),(21,[1,3,7]),\ldots,(50,[1,2,5,10,25])]$$

   - La lista de los números perfectos menores que 1000. Un número es perfecto si es igual a la suma de sus divisores (excluido él mismo). Por ejemplo, 6 es perfecto, pues 6=1+2+3.
   - La propiedad de ser un número primo.
   - La lista (infinita) de todos los números primos.
   - La lista de los números primos menores que 1000.
   - La cantidad de números primos que hay entre 200 y 500.
   - El primer número primo mayor que 6923.
   - El menor número primo a partir del cual hay 30 números consecutivos que no son primos.
   - La lista de los 5 primeros números *procèsables*. Un número natural n es *procèsable* si $n = pq$, siendo p, q primos y verificándose además que x es la suma de todos los primos entre p y q (ambos incluidos). Por ejemplo, 155 es *procèsable* pues $155 = 5 \cdot 31$ y $155 = 5 + 7 + 11 + 13 + 17 + 19 + 23 + 29 + 31$.
   - La descomposición en factores primos de un número, en una representación ilustrada por este ejemplo: `fprimos 1176 = [(2,3),(3,1),(7,2)]`, pues $1176 = 2^3 \cdot 3 \cdot 7^2$.

4. Estudia si puedes definir mediante `foldr` o `foldl`, en lugar de mediante recursión explícita, las siguientes funciones: `last`, `reverse`, `all`, `any`, `concat`, `minimum`, `map`, `filter`, `takeWhile`, `take`, `(++)`.

5. Programa, indicando los tipos, las siguientes variantes de `foldl` y `foldr`, que operan con listas no vacías y no usan valor acumulado inicial:

   - $foldr1\ \bigoplus\ [x_1,\ldots,x_n] = x_1 \bigoplus x_2 \bigoplus \ldots \bigoplus x_n$ (con $\bigoplus$ asociando por la derecha)
   - $foldl1\ \bigoplus\ [x_1,\ldots,x_n] = x_1 \bigoplus x_2 \bigoplus \ldots \bigoplus x_n$ (con $\bigoplus$ asociando por la izquierda)

6. Programa las siguientes funciones, teniendo en cuenta listas intensionales por si son de utilidad.

```
prefixes,suffixes,sublists,parts,perms:: [a] -> [[a]]
-- (prefixes xs) devuelve las lista de todos los prefijos de xs.
-- (suffixes xs) devuelve las lista de todos los sufijos de xs.
-- (sublists xs) devuelve las lista de todas las sublistas de xs.
-- (partss xs) devuelve las lista de todos las partes (subconjuntos) de xs.
-- (perms xs) devuelve la lista de todas las permutaciones de xs.
-- (inits xs) devuelve las lista de todos los segmentos iniciales de xs,
--  en orden de longitud creciente.

combinaciones,variaciones:: Int -> [a] -> [[a]]
-- (combinaciones n xs) devuelve la lista de todas las combinaciones de
--                      elementos de xs tomados de n en n
-- Algo similar para variaciones
sumandos:: Int -> [[Int]]
-- sumandos n devuelve la lista de todas las descomposiciones en sumandos positivos de n
-- Ejemplo: sumandos 3 = [[1,1,1],[1,2],[2,1],[3]]
-- O bien, si no queremos resultados que sean permutaciones unos de otros:
--   sumandos 3 = [[1,1,1],[1,2],[3]]
```

7. Programa el producto escalar de vectores y el producto de matrices, suponiento que los vectores se representan como `[Float]` y las matrices como listas de vectores.

8. Elimina las listas intensionales de las siguientes definiciones, usando *map, filter* y *concat*:

```
f n     = [x*x | x <- [1..n], mod x 2 == 0]
g n m   = [x+y | x <- [1..n], y <- [x..m]]
h p n m = [x+y | x <- [1..n], p (n-x), y <- [x..m]]
```

9. Elimina, reemplazándolas por funciones auxiliares no locales, las definiciones locales y la λ-abstracción de la definición siguiente:

```
f x y = map (\u -> (g u,g (u+1))) y
        where z = x * last y
              g u = (x+z)*u
```

10. Considera la función
```
f  n
   | mod n 2 == 0   = div n 2
   | otherwise      = 3*n + 1
```
La llamada *conjetura de Collatz* afirma que, para cualquier n, al iterar f a partir de n siempre se alcanza el valor 1.

    - Haz alguna comprobación de la conjetura *(usa la función iterate, por Dios bendito).*
    - Mecaniza un poquito la experimentación, programando por ejemplo las siguientes funciones:
        - $f'n$ = número de pasos que hay que iterar f desde n para llegar a 1.
        - $f''n$ = número de pasos que hay que iterar f desde n para llegar a 1, junto con la lista de los resultados intermedios.
        - $f'''n$ = lista de pares (i, k), con i creciente desde 1 a n y k el número de pasos requeridos para alcanzar 1 iterando f desde i.
    - Calcula el primer n que requiere más de 150 pasos de iteración de f para llegar a 1.
    - Calcula la lista de los números entre 1000 y 2000 que requieren más de 150 pasos de iteración de f para llegar a 1.
    - Calcula la lista de los números n que requieren más de n iteraciones de f para llegar a 1.

11. Considera la siguiente función:

```
fix:: (a -> a) -> a
fix f = f (fix f)
```

Con su ayuda, toda la recursión de un programa puede eliminarse (salvo la de la propia `fix`), expresando cada función definida recursivamente como punto fijo (o sea, como aplicación de `fix`) de una función de orden superior asociada que ya no es recursiva. Por ejemplo, la definición del factorial

```
fac n = if n==0 then 1 else n*fac (n-1)
```

podemos reemplazarla por

```
fac1  = fix facHO

-- facHO: como fac, pero cambiando la aparición recursiva de fac por un parámetro f

facHO f n = if n==0 then 1 else n*f (n-1)
```

    - Comprueba en algún ejemplo que `fac1` computa efectivamente el factorial
    - Examina el tipo de `facHO`
    - Haz a mano el cómputo de `fac1 3` para entender mejor lo que sucede
    - Aplica este método a otras funciones recursivas, como por ejemplo `length, ++, ...`
    - ¿Resultan las definiciones mediante `fix` mucho más ineficientes que las originales? *(Para una comparación justa, no debes usar las versiones primitivas de* `length`*,* `++`*, ..., sino versiones programadas por ti)*