Programación Funcional

Curso 2016/2017 – 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. 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 menores que 10^{10} que acaban 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 de los números primos menores que 1000 *(necesitarás previamente programar la propiedad de ser un número primo).*
   - La cantidad de números primos que hay entre 200 y 500.
   - El primer número primo mayor que 6923.
   - 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
   - El menor número primo a partir del cual hay 30 números consecutivos que no son primos.

2. Piensa y/o prueba en el intérprete cuáles de las siguientes expresiones tardarán poco (digamos centésimas o milésimas de segundos), regular (digamos décimas o segundos) o mucho (digamos toda una vida) en ser evaluadas. En el tercer caso, no te esperes toda tu vida, sino que estima cuánto va a tardar la evaluación o, al menos, interrumpe el cómputo.

   - `last [1..10^5]`
   - `last [1..10^7]`
   - `last [1..10^20]`
   - `head [1..10^20]`
   - `last [10^20..1]`
   - `head (tail [1..10^20])`
   - `length [1..10^20]`
   - `last (take (10^7) [1..10^20])`
   - `head (take (10^7) ([1..100] ++ [1..10^20]))`
   - `last (take 100 ([1..10^20] ++ [1..100]))`

- `last (drop 100 ([1..10^20] ++ [1..100]))`
- `head (drop (10^7) ([1..10^20] ++ [1..100]))`
- `[1..10^7]==[1..10^7]`
- `[1..10^20]==[1..10^20]`
- `[1..10^20]==[1..10^20+1]`
- `[1..10^20]==[2..10^20]`
- `head (reverse [1..10^7])`
- `last (reverse [1..10^7])`
- `reverse [1..10^20] == reverse [1..10^20+1]`

3. Programa las siguientes funciones, indicando sus tipos:

   De momento, prográmalas usando explícitamente recursión. Más adelante, revísalas para ver si en alguna de ellas puedes ocultar la recursión mediante funciones de orden superior. Muchas de estas funciones están en el `Prelude` (puedes usar `:t` o `:info` para descubrir si están en ámbito en el intérprete), por lo que te convendrá renombrarlas.

   `last xs` = último elemento de la lista no vacía `xs`
   `init xs` = todos menos el último elemento de la lista no vacía `xs`
   `initLast xs` = `(init xs,last xs)`
   `concat xss` = resultado de concatenar los elementos de la lista de listas `xss`
   `take n xs` = lista de los `n` primeros elementos de `xs`
   `drop n xs` = resultado de eliminar los `n` primeros elementos de `xs`
   `splitAt n xs` = `(take n xs, drop n xs)`
   `reverse xs` = inversa de la lista `xs`
   `nub xs` = resultado de eliminar los elementos repetidos de la lista `xs`
   `and bs` = resultado de hacer la conjunción de todos los elementos de `bs`
   `or bs` = resultado de hacer la disyunción de todos los elementos de `bs`
   `sum xs` = resultado de sumar todos los elementos de `xs`
   `product xs` = resultado de multiplicar todos los elementos de `xs`
   `mean xs` = media aritmética de los elementos de `xs`
   `lmedia xss` = longitud media de los elementos de la lista de listas `xss`
   `sort xs` = resultado de ordenar la lista `xs` (usa diferentes métodos)

4. 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.
   - `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

5. 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.

6. La población de India a final de 2014 era de 1267 millones de habitantes, con un crecimiento del 0.78 % respecto al año anterior. Los datos análogos para China fueron 1368 millones y 0.70 %, respectivamente. Suponiendo que los ritmos de crecimiento se mantienen constantes a lo largo de los años, calcula mediante Haskell lo siguiente:

   - Las poblaciones de ambos países en 2013 y en 2105.
   - Las poblaciones de ambos países en 2000 y 2030.
   - Una lista con la población de India en 2014 y años sucesivos, o sea de la forma `[1267,Pob_2015,Pob_2016,...]`. Lo mismo para China.
   - Una lista como la anterior, pero con cada población emparejada con el año, o sea, de la forma `[(2014,1267),(2015,Pob_2015),(2016,Pob_2016),...]`. Lo mismo para China.
   - Una lista con la diferencia de población entre China e India, en años desde 2014 en adelante.
   - Una lista como la anterior, pero con cada diferencia de población emparejada con el año.
   - El año en que India superará en población a China

7. 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 algunos ejemplos 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` claramente más ineficientes que las originales? *(Para una comparación justa, no debes usar las versiones primitivas de* `length`*,* `++`*, ..., sino versiones programadas por ti)*