Práctico 7 de Programación Funcional

UdelaR/FCien/CMat

xx al yy de octubre de 2016

1. (a) Probar que si `p, q::a->Bool`, entonces:
   `(filter p).(filter q) = filter (and p q)` donde
   `and p q x = (p x)&&(q x)`.
   Obs.: Recordar que `&&` es una función no estricta en el segundo argumento, pero sí en el primero.

   (b) Sea `p::a->Bool`. Probar que
   `(filter p).concat = concat.(map (filter p))`.

   (c) Probar que `map f xs = [f x | x<-xs]` para toda `f::a->b` y toda lista `xs::a`.

   (d) Probar que `filter p xs = [x | x<-xs, p x]` para toda `p::a->Bool` y toda `xs::[a]`.

   (e) Probar que `[e | Q, P] = concat [[e | P] | Q]` (`P, Q` son calificadores).

   (f) Probar que `[e | Q, x<-[d | P]] = [e{x:=d} | Q, P]` (`P, Q` son calificadores, `e{x:=d}` es el resultado de substituir en `e` la variable `x` por la expresión `d`).

2. Se definen

```
inits          ::  [a]->[[a]]
inits []       =   []
inits (x:xs)   =   []:(map (x:)(inits xs))        (1)
scanl          ::  (b->a->b)->b->[a]->[b]
scanl f e      =   (map (foldl f e)).inits
```

   Probar que

```
scanl f e []      =  [e]
scanl f e (x:xs)  =  e:(scanl f (f e x) xs)
```

   Observar que esta propiedad permite definir `scanl` y probar que la definición que se obtiene de esta forma es más eficiente que dada en la tabla (1).

3. Probar que `map` es un functor (en la categoría de los tipos del Haskell y las funciones Haskell-computables).

(a) Probar que `head : map -> Id` es una transformación natural del functor `map` en el functor `Id`.

(b) Probar que `tail : map -> map` es una transformación natural de `map` en `map`.

(c) Probar que `concat : map.map -> map` es una transformación natural de `map.map` en `map`.

4. Considerar la definición

```
approx               ::  Integer -> [a] -> [a]
approx (n+1) []      =   []
approx (n+1) (x:xs)  =   x:(approx n xs)
```

(a) Probar que $\forall$`xs::[a]` $(\texttt{xs})_n :=$ `approx n xs` es monótona.

(b) Probar que $\forall$`xs::[a]` $\lim_{n\to+\infty} \texttt{xs}_n =$ `xs`.

(c) Probar que si $\forall n{\in}\mathbb{N}\ \exists m{\in}\mathbb{N}$ `approx n xs` $\sqsubseteq_{\texttt{[a]}}$ `approx m ys` entonces `xs` $\sqsubseteq_{\texttt{a}}$ `ys` para todo par de listas `xs, ys :: [a]`.

(d) Concluir que `xs = ys` si y sólo si ($\forall n{\in}\mathbb{N}\ \exists m{\in}\mathbb{N}$ `approx n xs` $\sqsubseteq_{\texttt{[a]}}$ `approx m ys`) $\wedge$ ($\forall n{\in}\mathbb{N}\ \exists m{\in}\mathbb{N}$ `approx n ys` $\sqsubseteq_{\texttt{[a]}}$ `approx m xs`).

(e) Se considera la definición

```
take               ::  Integer -> [a] -> [a]
take 0 xs          =   []
take (n+1) []      =   []
take (n+1) (x:xs)  =   x:(take n xs)
```

Probar que $\forall$`xs, ys::[a] xs=ys` si y sólo si $\forall n{\in}\mathbb{N}$ `take n xs = take n ys`.

(f) Se considera la definición

```
(!!)              ::  [a] -> Integer -> a
(x:xs) !! 0       =   x
(x:xs) !! (n+1)   =   xs !! n
```

¿Es cierto que $\forall$`xs, ys::[a]` finitas, si $\forall n{\in}\mathbb{N}$ `xs!!n = ys!!n` entonces `xs = ys`? Probarlo o dar un contraejemplo.

¿Es cierto lo anterior para listas cualesquiera? Probarlo o dar un contraejemplo.

5. Probar que no es posible definir una función de Haskell que ordene en sentido creciente a las listas infinitas del tipo `[Integer]`.

6. (a) Probar que $\forall$`f::a -> a` $\forall n{\in}\mathbb{N}$ `approx n.map f = map f.approx n`.

(b) Se consideran las definiciones

```
nats    ::  [Integer]
nats    =   0:map (+1) nats
from    ::  Integer -> [Integer]
from x  =   x:from (x+1)
```

Probar que `nats = from 0`.

7. Probar que `iterate (+a) b = [(i*a)+b | i <- [0..]]`

8. Se considera la definición

   ```
   fibs = 0:1:[x+y | (x,y) <- zip fibs (tail fibs)].
   ```

   Probar que `fibs = map fib [0..]` donde `fib` es una expresión que calcula la sucesión de Fibonacci.

9. *Torres de Hanoi.* El juego está conformado por tres astas y una pila de discos de diámetros dos a dos distintos. En el momento de incicializar la partida, la pila de discos está en el asta 1, de forma tal que ninguno de ellos se apoya sobre otro de diámetro menor. El objetivo del juego es mover los discos del asta 1 al asta 3 respetando las siguientes reglas:

   - Se puede mover sólo un disco por vez.
   - No se puede poner un disco sobre otro más chico.
   - En todo momento cada uno de los discos debe de estar en una de las astas.

   Se pide:

   (a) Definir `hanoi :: Integer -> [(Integer, Integer)]` que recibe el número de discos y devuelve una estrategia para el juego, según la siguiente codificación de las movidas:

      i. Cada par en la lista `hanoi n` representa una movida, e.g.: el par `(2,1)` significa *mover el disco superior de la pila del asta 2 y colocarlo encima de la pila del asta 1.*

      ii. Las movidas han de ejecutarse de izquierda a derecha en el orden en el que aparecen en `hanoi n`.

   (b) Probar `length (hanoi n) = 2*length (hanoi (n-1)) + 1` y encontrar una fórmula explícita para `length (hanoi n)`. ¿Qué orden asintótico de infinito tiene el programa hallado?

10. La Criba de Eratóstenes es un método para generar la lista de todos los números primos. La idea es comenzar con la lista `[2..]` y después aplicar repetidamente el siguiente proceso:

    i. Marcar el primer elemento de la lista como primo

    ii. Borrar de la lista todos los múltiplos de p.

    iii. Aplicar el método a la cola (tail) de la lista resultante.

    Definir `sieve :: [Integer] -> [Integer]` que calculada sobre una lista `p:xs` calcule las operaciones correspondientes al paso ii.

    Definir `primes` que calcule la lista de todos los números primos utilizando `iterate` y `sieve`.

11. El objetivo de este ejercicio es implementar el juego "piedra-papel-tijeras" visto en el teórico, haciendo jugar a las estrategias "recíproca" y "estadística".

    (a) Para definir la función `random` necesaria a los efectos de implementar la estrategia estadística, definir la siguiente función:

    ```
    randoms     ::     [Integer]
    randoms     =      iterate f semilla
              where    f x = mod (a * x + c) m
    ```

    Los identificadores `semilla, a, c` y `m` son parámetros de cuya elección depende la eficacia de la función para producir números pseudo-aleatorios.

    Elegimos los siguientes valores: $a = 69069$, $c = 1$, $m = 2^{32}$. Estos valores corresponden al generador pseudo-aleatorio usado en las máquinas de arquitectura "VAX". Como semilla elegir cualquier valor de tipo `Integer`.

    Completar la definición de la estrategia estadística, definiendo la función `choose`.

    (b) Implementar el juego mediante la función `match` vista en clase. Implementar la función `rounds` y la estrategia estadística según la primera versión vista en clase (i.e.: la que hace que el juego demore tiempo $\Omega(n^2)$ si n es el largo de la partida). Hacer jugar las estrategias implementadas una contra otra. Compilar el programa con las opciones `-prof -fprof-auto -rtsopts` y correr el ejecutable con los parámetros `+RTS -p`. Comparar cuantos recursos utiliza cada versión.

    (c) Implementar el juego mediante la función `match` definiendo `rounds` según la segunda versión vista en clase (i.e.: la que usa a la función `scanl` para producir la lista de las distribuciones de `Paper, Rock` y `Scissors` en las jugadas del contrincante y corre en tiempo lineal). Hacer jugar las estrategias implementadas una contra otra. Compilar el programa con las opciones `-prof -fprof-auto -rtsopts` y correr el ejecutable con los parámetros `+RTS -p`. Comparar cuantos recursos utiliza cada versión.