Programación declarativa **(29 de noviembre de 2010)**

Apellidos:

Nombre: **Grupo:**

Ejercicio 1 [2 puntos] La función `unzip` recibe una lista de pares xs y devuelve un par de listas de la misma longitud (as, bs) tales que $xs =$ `zip` $as\ bs$.
Por ejemplo:

```
unzip [(1,2),(4,6),(2,5)] ⇝ ([1,4,2],[2,6,5])
```

1. ¿Cuál es el tipo más general para la función `unzip`?
2. Escribe una definición *recursiva* de la función `unzip`.
3. Definimos la propiedad

   ```
   prop ::  [Int] -> [Int] -> Bool
   prop xs ys = unzip (zip xs ys) == (xs,ys)
   ```

   ¿Qué ocurrirá si evaluamos la expresión `quickCheck prop` ?
4. Definimos la función

   ```
   f = sum .  fst .  unzip
   ```

   Describe el comportamiento de `f` y el tipo más general para dicha función.

Ejercicio 2 [2 puntos] Se define por recursión la función:

```
existe p y (x:xs) = p y x || existe p y xs -- e1
existe _ _ _ = False                       -- e2
```

1. Determina el tipo más general para la función `existe`.
2. Escribe cómo evalúa Haskell paso a paso las expresiones

   ```
   existe (==) 'a' "ole"
   existe (<=) 2 [1..]
   ```

   Nota: La definición del operador `||` en el Preludio de Haskell es:

   ```
   False || x = x   -- o1
   True || x = True -- o2
   ```
3. Escribe una definición *no recursiva* para la función `existe`.
4. ¿Qué responde Haskell al evaluar la siguiente expresión?

   ```
   map (existe (==) 2) [[i | i <- [0..x]] | x <- [0..2]]
   ```

Ejercicio 3 [3 puntos] Se define el tipo de datos `Arbol a`

```
data Arbol a = Hoja a | Nodo a (Arbol a) (Arbol a)
```

para representar árboles binarios que contienen elementos de tipo `a`.

1. Define en Haskell la función `ramas :: Arbol a -> [[a]]` tal que `(ramas x)` devuelve las ramas del árbol x. Por ejemplo:

   ```
   arbol1 = (Nodo 2 (Hoja 3) (Nodo 5 (Hoja 7) (Hoja 0)))
   arbol2 = (Nodo 7 (Hoja 3) (Hoja 7))
   ramas arbol1 ⇝ [[2,3],[2,5,7],[2,5,0]]
   ramas arbol2 ⇝ [[7,3],[7,7]]
   ```

2. Define en Haskell la función:

   ```
   sub :: Eq a => Arbol a -> Arbol a -> Bool
   ```

   tal que `(sub x y)` se verifica si todos los elementos del árbol x aparecen en en árbol y. Por ejemplo:

   ```
   sub arbol2 arbol1 ⇝ True
   sub arbol1 arbol2 ⇝ False
   ```

Ejercicio 4 [3 puntos]

1. Define en Haskell la función `subset :: Int -> [a] -> [[a]]` tal que `(subset x xs)` devuelve los subconjuntos de xs de tamaño x.

   ```
   subset 2 "abc" ⇝ ["ab","ac","bc"]
   ```

2. Define en Haskell la función

   ```
   sss :: [Int] -> Int -> [[Int]]
   ```

   tal que `(sss xs x)` devuelve la lista de todos los subconjuntos de xs cuya suma es x. Por ejemplo:

   ```
   sss [1,2,-3,4,5] 2 ⇝ [[2],[-3,5],[1,-3,4]]
   sss [1,2,-3,4,5] (-5) ⇝ []
   ```

3. Define un nuevo tipo de dato `Sol` , haz dicho tipo de dato instancia de la clase `Show`, y define una nueva versión de la función `sss`

   ```
   sssMod :: [Int] -> Int -> Sol
   ```

   de manera que se produzca el siguiente comportamiento:

   ```
   sssMod [1,2,-3,4,5] 2 ⇝ [[2],[-3,5],[1,-3,4]]
   sssMod [1,2,-3,4,5] (-5) ⇝ No hay solución
   ```
