Introducción a la programación

Práctica 5: Recursión sobre listas

Ej 1.4

Ejercicio 1. Definir las siguientes funciones sobre listas

`problema reverso` (s: $seq\langle\mathbb{Z}\rangle$) : $seq\langle\mathbb{Z}\rangle$ {
  `requiere:` { True }
  `asegura:` { $resultado$ tiene los mismos elementos que s pero en orden inverso}
}

Ej 1.4

```
reverso :: [t] -> [t]
reverso []     = []
reverso (x:xs) = reverso xs ++ [x]
```

Ej 2.3

Ejercicio 2. Definir las siguientes funciones sobre listas

problema `todosDistintos` (s: $seq\langle\mathbb{Z}\rangle$) : Bool $\{$
  `requiere:` $\{$ True $\}$
  `asegura:` $\{ resultado = false \leftrightarrow (\exists i, j : \mathbb{Z})(0 \leq i < |s| \land 0 \leq j < |s| \land i \neq j \land s[i] = s[j])\}$
$\}$

Ej 2.3

```
todosDistintos :: (Eq t) => [t] -> Bool
todosDistintos [] = True
todosDistintos (x:xs)
               | pertenece x xs = False
               | otherwise = todosDistintos xs
```

Ej 3.3

Ejercicio 3. Definir las siguientes funciones sobre listas de enteros

problema maximo (s: $seq\langle\mathbb{Z}\rangle$) : $\mathbb{Z}$ {
  requiere: { $|s| > 0$ }
  asegura: {
    $resultado \in s \land (\forall i : \mathbb{Z})(0 \leq i < |s| \rightarrow resultado \geq s[i]\}$
}

Ej 3.3

```
maximo :: [Integer] -> Integer
maximo [x] = x
maximo (x:y:xs) | x > y = maximo (x:xs)
                | otherwise = maximo (y:xs)
```

Ej 3.7

Ejercicio 3. Definir las siguientes funciones sobre listas de enteros

`problema pares` (s: $seq\langle\mathbb{Z}\rangle$) : $seq\langle ent\rangle$ {
  `requiere:` { True }
  `asegura:` { $resultado$ sólo tiene los elementos pares de s en el orden dado, respetando las repeticiones.}
}

Ej 3.7

```
pares :: [Integer] -> [Integer]
pares [] = []
pares (x:xs) | mod x 2 == 0 =  x : pares xs
             | otherwise = pares xs
```

Ej 3.9

Ejercicio 3. Definir las siguientes funciones sobre listas de enteros

`problema ordenar` (s: $seq\langle\mathbb{Z}\rangle$) : $seq\langle ent\rangle$ {
  `requiere:` { True }
  `asegura:` { $resultado$ contiene los elementos de s ordenados de forma creciente}
}

Ej 3.9

```
ordenar :: [Integer] -> [Integer]
ordenar [] = []
ordenar xs = min : ordenar (sacar xs min)
    where min = minimo xs

sacar :: [Integer] -> Integer -> [Integer]
sacar [y] x = []
sacar (y:ys) x | x == y = ys
               | otherwise = y : sacar ys x

minimo :: [Integer] -> Integer
minimo [x] = x
minimo (x:y:xs) = if (x < y) then
        minimo (x:xs) else minimo (y:xs)
```

Ej 4.4

Ejercicio 4. Definir las siguientes funciones sobre listas de caracteres, interpretando una palabra como una secuencia de caracteres sin blancos:

`problema aplanar` (s: $seq\langle seq\langle Char\rangle\rangle$) : $seq\langle Char\rangle$ {
  `requiere:` { True }
  `asegura:` {$resultado$ es la contatenación de las palabras de s}
}

Ej 4.4

```
aplanar :: [[Char]] -> [Char]
aplanar [] = []
aplanar [x] = x
aplanar (x:xs) = x ++ [' '] ++ aplanar xs
```

Ej 5.1

Ejercicio 5. Definir las siguientes funciones sobre listas:
`nat2bin :: Integer -> [Integer]`, que recibe un número no negativo y lo transforma en una lista de bits correspondiente a su representación binaria.
Por ejemplo `nat2bin 8` devuelve `[1, 0, 0, 0]`.

Ej 5.1

```
nat2bin :: Integer -> [Integer]
nat2bin 0 = [0]
nat2bin 1 = [1]
nat2bin x = nat2bin (div x 2) ++ [mod x 2]
```

Ej 6.2

Ejercicio 6. Definir las siguientes funciones sobre conjuntos:
`partes :: Integer -> Set (Set Integer)`, que genere todos los subconjuntos del conjunto {1,2,..n}.
Por ejemplo `partes 2` devuelve `[[], [1], [2], [1, 2]]`

Ej 6.2

```
partes :: Int -> [[Int]]
partes 0 = [[]]
partes x = agregarATodos x partesMasChico
                  ++ partesMasChico
    where partesMasChico = partes (x-1)

agregarATodos :: Int -> [[Int]] -> [[Int]]
agregarATodos x [] = []
agregarATodos x (y:ys) = (x:y) :
    agregarATodos x ys
```