Ejercicios de programación funcional con Haskell

José A. Alonso Jiménez

Grupo de Lógica Computacional
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, 8 de Agosto de 2007 (Versión de 20 de septiembre de 2007)

Índice general

Capítulo 1

Introducción a la programación funcional

Contenido

1.1. Factorial

Ejercicio 1.1. *Definir la función* `factorial` *tal que* `factorial n` *es el factorial de* `n`*. Por ejemplo,*

```
factorial 4 ⇝24
```

Solución: Vamos a presentar distintas definiciones.

1. Primera definición: Con condicionales:

```
fact1 :: Integer -> Integer
fact1 n =  if n == 0 then 1
                     else n * fact1 (n-1)
```

2. Segunda definición: Mediante *guardas*:

```
fact2 :: Integer -> Integer
fact2 n
    | n == 0    = 1
    | otherwise = n * fact2 (n-1)
```

3. Tercera definición: Mediante *patrones*:

```
fact3 :: Integer -> Integer
fact3 0 =  1
fact3 n =  n * fact3 (n-1)
```

4. Cuarta definición: Restricción del dominio mediante guardas

```
fact4 :: Integer -> Integer
fact4 n
    | n == 0  = 1
    | n >= 1  = n * fact4 (n-1)
```

5. Quinta definición: Restricción del dominio mediante patrones:

```
fact5 :: Integer -> Integer
fact5 0     =  1
fact5 (n+1) =  (n+1) * fact5 n
```

6. Sexta definición: Mediante *predefinidas*

```
fact6 :: Integer -> Integer
fact6 n =  product [1..n]
```

7. Séptima definición: Mediante plegado:

```
fact7 :: Integer -> Integer
fact7 n = foldr (*) 1 [1..n]
```

Se pueden comprobar todas las definiciones con

```
Factorial> [f 4 | f <- [fact1,fact2,fact3,fact4,fact5,fact6,fact7]]
[24,24,24,24,24,24,24]
```

Las definiciones son equivalentes sobre los números naturales:

```
prop_equivalencia :: Integer -> Property
prop_equivalencia x =
    x >= 0 ==> (fact2 x == fact1 x &&
                fact3 x == fact1 x &&
                fact4 x == fact1 x &&
                fact5 x == fact1 x &&
                fact6 x == fact1 x &&
                fact7 x == fact1 x)
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

La definición más elegante es la quinta y es a la que nos referimos como `factorial`

```
factorial = fact5
```

1.2. Número de combinaciones

Ejercicio 1.2. *Definir la función* `comb` *tal que* `comb n k` *es el número de combinaciones de* `n` *elementos tomados de* `k` *en* `k`*; es decir,*

$$\binom{n}{m} = \frac{n!}{k!\,(n-k)!}$$

Por ejemplo.

```
comb 6 2 ⇝ 15
```

Solución:

```
comb n k = (factorial n) `div` ((factorial k) * (factorial (n-k)))
```

1.3. Comprobación de número impar

Ejercicio 1.3. *Definir la función* `impar` *tal que* `impar x` *se verifica si el número* `x` *es impar. Por ejemplo,*

```
impar 7  ⇝  True
impar 6  ⇝  False
```

Solución: Presentamos distintas definiciones:

1. Usando la predefinida `odd`

```
impar1 :: Integer -> Bool
impar1 = odd
```

2. Usando las predefinidas `not` y `even`:

```
impar2 :: Integer -> Bool
impar2 x = not (even x)
```

3. Usando las predefinidas `not`, `even` y `(.)`:

```
impar3 :: Integer -> Bool
impar3 = not . even
```

4. Por recursión:

```
impar4 :: Integer -> Bool
impar4 x | x > 0     = impar4_aux x
         | otherwise = impar4_aux (-x)
         where impar4_aux 0     = False
               impar4_aux 1     = True
               impar4_aux (n+2) = impar4_aux n
```

Las definiciones son equivalentes:

```
prop_equivalencia :: Integer -> Bool
prop_equivalencia x =
    impar2 x == impar1 x &&
    impar3 x == impar1 x &&
    impar4 x == impar1 x
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

1.4. Cuadrado

Ejercicio 1.4. *Definir la función* `cuadrado` *tal que* `cuadrado x` *es el cuadrado del número* `x`. *Por ejemplo,*

```
cuadrado 3  ⇝  9
```

Solución: Presentamos distintas definiciones:

1. Mediante `(*)`

```
cuadrado_1 :: Num a => a -> a
cuadrado_1 x = x*x
```

2. Mediante `(^)`

```
cuadrado_2 :: Num a => a -> a
cuadrado_2 x = x^2
```

3. Mediante secciones:

```
cuadrado_3 :: Num a => a -> a
cuadrado_3 = (^2)
```

4. Usaremos como `cuadrado` la primera

```
cuadrado = cuadrado_1
```

Las definiciones son equivalentes:

```
prop_equivalencia :: Int -> Bool
prop_equivalencia x =
    cuadrado_2 x == cuadrado_1 x &&
    cuadrado_3 x == cuadrado_1 x
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

1.5. Suma de cuadrados

Ejercicio 1.5. *Definir la función* `suma_de_cuadrados` *tal que* `suma_de_cuadrados l` *es la suma de los cuadrados de los elementos de la lista* `l`*. Por ejemplo,*

```
suma_de_cuadrados [1,2,3]  ⇝  14
```

Solución: Presentamos distintas definiciones:

1. Con `sum`, `map` y `cuadrado`:

```
suma_de_cuadrados_1 :: [Integer] -> Integer
suma_de_cuadrados_1 l = sum (map cuadrado l)
```

2. Con `sum` y listas intnsionales:

```
suma_de_cuadrados_2 :: [Integer] -> Integer
suma_de_cuadrados_2 l = sum [x*x | x <- l]
```

3. Con `sum`, `map` y `lambda`:

```
suma_de_cuadrados_3 :: [Integer] -> Integer
suma_de_cuadrados_3 l = sum (map (\x -> x*x) l)
```

4. Por recursión:

```
suma_de_cuadrados_4 :: [Integer] -> Integer
suma_de_cuadrados_4 []     = 0
suma_de_cuadrados_4 (x:xs) = x*x + suma_de_cuadrados_4 xs
```

Las definiciones son equivalentes:

```
prop_equivalencia :: [Integer] -> Bool
prop_equivalencia xs =
    suma_de_cuadrados_2 xs == suma_de_cuadrados_1 xs &&
    suma_de_cuadrados_3 xs == suma_de_cuadrados_1 xs &&
    suma_de_cuadrados_4 xs == suma_de_cuadrados_1 xs
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

1.6. Raices de ecuaciones de segundo grado

Ejercicio 1.6. *Definir la función* `raices` *tal que* `raices a b c` *es la lista de las raices de la ecuación* $ax^2 + bc + c = 0$. *Por ejemplo,*

```
raices 1 3 2  ⇝  [-1.0,-2.0]
```

Solución: Presentamos distintas definiciones:

1. Definición directa:

```
raices_1 :: Double -> Double -> Double -> [Double]
raices_1 a b c = [ (-b+sqrt(b*b-4*a*c))/(2*a),
                   (-b-sqrt(b*b-4*a*c))/(2*a) ]
```

2. Con entornos locales

```
raices_2 :: Double -> Double -> Double -> [Double]
raices_2 a b c =
    [(-b+d)/n, (-b-d)/n]
    where d  = sqrt(b*b-4*a*c)
          n = 2*a
```

La segunda es mejor en legibilidad y en eficiencia:

```
Main> :set +s
Main> raices_1 1 3 2
[-1.0,-2.0]
(134 reductions, 242 cells)
Main> raices_2 1 3 2
[-1.0,-2.0]
(104 reductions, 183 cells)
```

1.7. Valor absoluto

Ejercicio 1.7. *Redefinir la función* `abs` *tal que* `abs x` *es el valor absoluto de* `x`. *Por ejemplo,*

```
abs (-3) ⇝ 3
abs 3    ⇝ 3
```

Solución: Presentamos distintas definiciones:

1. Con condicionales:

```
n_abs_1 :: (Num a, Ord a) => a -> a
n_abs_1 x = if x>0 then x else (-x)
```

2. Con guardas:

```
n_abs_2 :: (Num a, Ord a) => a -> a
n_abs_2 x | x>0       = x
          | otherwise = -x
```

Las definiciones son equivalentes

```
prop_equivalencia :: Int -> Bool
prop_equivalencia x =
    n_abs_1 x == abs x &&
    n_abs_2 x == abs x
```

1.8. Signo

Ejercicio 1.8. *Redefinir la función* `signum` *tal que* `signum x` *es -1 si* `x` *es negativo, 0 si* `x` *es cero y 1 si* `x` *es positivo. Por ejemplo,*

```
signum 7    ⇝ 1
signum 0    ⇝ 0
signum (-4) ⇝ -1
```

Solución:

```
n_signum x | x > 0      = 1
           | x == 0     = 0
           | otherwise  = -1
```

Las definiciones son equivalentes:

```
prop_equivalencia :: Int -> Bool
prop_equivalencia x =
    n_signum x == signum x
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

1.9. Conjunción

Ejercicio 1.9. *Redefinir la función* `&&` *tal que* `x && y` *es la conjunción de* `x` *e* `y` *Por ejemplo,*

```
True && False ⇝ False
```

Solución:

```
(&&&) :: Bool -> Bool -> Bool
False &&& x   = False
True  &&& x   = x
```

Las definiciones son equivalentes:

```
prop_equivalencia x y =
    (x &&& y) == (x && y)
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

1.10. Anterior de un número natural

Ejercicio 1.10. *Definir la función* `anterior` *tal que* `anterior x` *es el anterior del número natural* `x`*. Por ejemplo,*

```
anterior 3 ⇝ 2
anterior 0 ⇝ Program error: pattern match failure: anterior 0
```

Solución: Presentamos distintas definiciones:

1. Con patrones:

   ```
   anterior_1 :: Int -> Int
   anterior_1 (n+1) = n
   ```

2. Con guardas:

   ```
   anterior_2 :: Int -> Int
   anterior_2 n | n>0 = n-1
   ```

   Las definiciones son equivalentes sobre los números naturales:

```
prop_equivalencia :: Int -> Property
prop_equivalencia n =
    n>0 ==> anterior_1 n == anterior_2 n
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

1.11. Potencia

Ejercicio 1.11. *Redefinir la función* `potencia` *tal que* `potencia x y` *es* x^y. *Por ejemplo,*

```
potencia 2 4   ⇝ 16
potencia 3.1 2 ⇝ 9.61
```

Solución: Presentamos distintas definiciones:

1. Por patrones:

```
potencia_1 :: Num a => a -> Int -> a
potencia_1 x 0     = 1
potencia_1 x (n+1) = x * (potencia_1 x n)
```

2. Por condicionales:

```
potencia_2 :: Num a => a -> Int -> a
potencia_2 x n = if n==0 then 1
                 else x * potencia_2 x (n-1)
```

3. Definición eficiente:

```
potencia_3 :: Num a => a -> Int -> a
potencia_3 x 0           = 1
potencia_3 x n  | n > 0  = f x (n-1) x
                where f _ 0 y = y
                      f x n y = g x n
                          where g x n | even n    = g (x*x) (n`quot`2)
                                      | otherwise = f x (n-1) (x*y)
```

Las definiciones son equivalentes:

```
prop_equivalencia :: Int -> Int -> Property
prop_equivalencia x n =
    n >= 0 ==>
    (potencia_1 x n == x^n &&
     potencia_2 x n == x^n &&
     potencia_3 x n == x^n)
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

1.12. Función identidad

Ejercicio 1.12. *Redefinir la función* `id` *tal que* `id x` *es* `x`*. Por ejemplo,*

```
id 3 ⇝ 3
```

Solución: La definición es

```
n_id  :: a -> a
n_id x = x
```

Capítulo 2

Números y funciones

Contenido

2.1. Casi igual

Ejercicio 2.1. *Definir el operador* `~=` *tal que* `x ~= y` *se verifica si* $|x - y| < 0{,}0001$. *Por ejemplo,*

```
3.00001 ~= 3.00002 ⇝ True
3.1 ~= 3.2         ⇝ False
```

Solución:

```
infix 4 ~=
(~=)   :: Float -> Float -> Bool
x ~= y  = abs(x-y) < 0.0001
```

2.2. Siguiente de un número

Ejercicio 2.2. *Definir la función* `siguiente` *tal que* `siguiente x` *sea el siguiente del número entero* `x`. *Por ejemplo,*

```
siguiente 3 ⇝ 4
```

Solución: Presentamos distintas definiciones:

1. Mediante sección:

   ```
   siguiente_1 :: Integer -> Integer
   siguiente_1 = (+1)
   ```

2. Mediante instanciación parcial:

   ```
   siguiente_2 :: Integer -> Integer
   siguiente_2 = (+) 1
   ```

3. Usaremos como `siguiente` la primera

```
siguiente = siguiente_1
```

Las definiciones son equivalentes:

```
prop_equivalencia :: Integer -> Bool
prop_equivalencia x =
    siguiente_1 x == siguiente_2 x
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

2.3. Doble

Ejercicio 2.3. *Definir la función* `doble` *tal que* `doble x` *es el doble de* `x`*. Por ejemplo,*

```
doble 3 ⇝ 6
```

Solución: Se presentan distintas definiciones:

1. Definición ecuacional:

```
doble_1 :: Num a => a -> a
doble_1 x = 2*x
```

2. Definición con instanciación parcial:

```
doble_2 :: Num a => a -> a
doble_2 = ((*) 2)
```

3. Definición con secciones:

```
doble_3 :: Num a => a -> a
doble_3 = (2*)
```

Las definiciones son equivalentes:

```
prop_equivalencia :: Int -> Bool
prop_equivalencia x =
    doble_2 x == doble_1 x &&
    doble_3 x == doble_1 x
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

2.4. Mitad

Ejercicio 2.4. *Definir la función* `mitad` *tal que* `mitad x` *es la mitad de* `x`*. Por ejemplo,*

```
mitad 6 ⇝ 3.0
mitad 5 ⇝ 2.5
```

Solución: Se presentan distintas definiciones:

1. Definición ecuacional:

```
mitad_1 :: Double -> Double
mitad_1 x = x/2
```

2. Definición con instanciación parcial:

```
mitad_2 :: Double -> Double
mitad_2 = (flip (/) 2)
```

3. Definición con secciones:

```
mitad_3 :: Double -> Double
mitad_3 = (/2)
```

Las definiciones son equivalentes para lon números no nulos:

```
prop_equivalencia :: Double -> Bool
prop_equivalencia x =
    mitad_2 x == mitad_1 x &&
    mitad_3 x == mitad_1 x
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

2.5. Inverso

Ejercicio 2.5. *Definir la función* `inverso` *tal que* `inverso x` *es el inverso de* `x`*. Por ejemplo,*

```
inverso 2 ⇝ 0.5
```

Solución: Se presentan distintas definiciones:

1. Definición ecuacional:

```
inverso_1 :: Double -> Double
inverso_1 x = 1/x
```

2. Definición con instanciación parcial:

```
inverso_2 :: Double -> Double
inverso_2 = ((/) 1)
```

3. Definición con secciones:

```
inverso_3 :: Double -> Double
inverso_3 = (1/)
```

Las definiciones son equivalentes para lon números no nulos:

```
prop_equivalencia :: Double -> Property
prop_equivalencia x =
    x /= 0 ==>
    (inverso_2 x == inverso_1 x &&
     inverso_3 x == inverso_1 x)
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

2.6. Potencia de dos

Ejercicio 2.6. *Definir la función* `dosElevadoA` *tal que* `dosElevadoA x` *es* 2^x*. Por ejemplo,*

```
dosElevadoA 3 ⇝ 8
```

Solución: Se presentan distintas definiciones:

1. Definición ecuacional:

```
dosElevadoA_1 :: Int -> Int
dosElevadoA_1 x = 2^x
```

2. Definición con instanciación parcial:

```
dosElevadoA_2 :: Int -> Int
dosElevadoA_2 = ((^) 2)
```

3. Definición con secciones:

```
dosElevadoA_3 :: Int -> Int
dosElevadoA_3 = (2^)
```

Las definiciones son equivalentes:

```
prop_equivalencia :: Int -> Property
prop_equivalencia x =
    x >= 0 ==>
      (dosElevadoA_2 x == dosElevadoA_1 x &&
       dosElevadoA_3 x == dosElevadoA_1 x)
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

2.7. Reconocimiento de números positivos

Ejercicio 2.7. *Definir la función* `esPositivo` *tal que* `esPositivo` *se verifica si* `x` *es positivo. Por ejemplo,*

```
esPositivo 3    ⇝ True
esPositivo (-3) ⇝ False
```

Solución: Se presentan distintas definiciones:

1. Definición ecuacional:

```
esPositivo_1 :: Int -> Bool
esPositivo_1 x = x>0
```

2. Definición con instanciación parcial:

```
esPositivo_2 :: Int -> Bool
esPositivo_2 = (flip (>) 0)
```

3. Definición con secciones:

```
esPositivo_3 :: Int -> Bool
esPositivo_3 = (>0)
```

Las definiciones son equivalentes:

```
prop_equivalencia :: Int -> Bool
prop_equivalencia x =
    esPositivo_2 x == esPositivo_1 x &&
    esPositivo_3 x == esPositivo_1 x
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

2.8. Aplicación de una función a los elementos de una lista

Ejercicio 2.8. *Redefinir la función* `map` *tal que* `map f l` *es la lista obtenida aplicando* `f` *a cada elemento de* `l`*. Por ejemplo,*

```
map (2*) [1,2,3] ⇝ [2,4,6]
```

Solución: Presentamos distintas definiciones:

1. Definición recursiva:

```
n_map_1 :: (a -> b) -> [a] -> [b]
n_map_1 f []     = []
n_map_1 f (x:xs) = f x : n_map_1 f xs
```

2. Con listas intensionales:

```
n_map_2 :: (a -> b) -> [a] -> [b]
n_map_2 f xs = [ f x | x <- xs ]
```

Las definiciones son equivalentes:

```
prop_equivalencia :: [Int] -> Bool
prop_equivalencia xs =
    n_map_1 (*2) xs == map (*2) xs &&
    n_map_2 (*2) xs == map (*2) xs
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

2.9. Filtrado mediante una propiedad

Ejercicio 2.9. *Redefinir la función* `filter` *tal que* `filter p l` *es la lista de los elementos de* `l` *que cumplen la propiedad* `p`*. Por ejemplo,*

```
filter even [1,3,5,4,2,6,1] ⇝ [4,2,6]
filter (>3) [1,3,5,4,2,6,1] ⇝ [5,4,6]
```

Solución: Presentamos distintas definiciones:

1. Definición por recursión:

```
n_filter_1 :: (a -> Bool) -> [a] -> [a]
n_filter_1 p []                = []
n_filter_1 p (x:xs) | p x      = x : n_filter_1 p xs
                    | otherwise = n_filter_1 p xs
```

2. Definición con listas intensionales:

```
n_filter_2 :: (a -> Bool) -> [a] -> [a]
n_filter_2 p xs = [ x | x <- xs, p x ]
```

Las definiciones son equivalentes cuando la propiedad es `even`:

```
prop_equivalencia :: [Int] -> Bool
prop_equivalencia xs =
    n_filter_1 even xs == filter even xs &&
    n_filter_2 even xs == filter even xs
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

2.10. Suma de los elementos de una lista

Ejercicio 2.10. *Redefinir la función* `sum` *tal que* `sum l` *es la suma de los elementos de* `l`*. Por ejemplo,*

```
n_sum [1,3,6] ⇝ 10
```

Solución: Presentamos distintas definiciones:

1. Definición recursiva:

```
n_sum_1 :: Num a => [a] -> a
n_sum_1 []     = 0
n_sum_1 (x:xs) = x + n_sum_1 xs
```

2. Definición con plegado:

```
n_sum_2 :: Num a => [a] -> a
n_sum_2 = foldr (+) 0
```

Las definiciones son equivalentes:

```
prop_equivalencia :: [Int] -> Bool
prop_equivalencia xs =
    n_sum_1 xs == sum xs &&
    n_sum_2 xs == sum xs
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

2.11. Producto de los elementos de una lista

Ejercicio 2.11. *Redefinir la función* `product` *tal que* `product l` *es el producto de los elementos de* `l`*. Por ejemplo,*

```
product [2,3,5] ⇝ 30
```

Solución: Presentamos distintas definiciones:

1. Definición recursiva

```
n_product_1 :: Num a => [a] -> a
n_product_1 []     = 1
n_product_1 (x:xs) = x * n_product_1 xs
```

2. Definición con plegado:

```
n_product_2 :: Num a => [a] -> a
n_product_2 = foldr (*) 1
```

Las definiciones son equivalentes:

```
prop_equivalencia :: [Int] -> Bool
prop_equivalencia xs =
    n_product_1 xs == product xs &&
    n_product_2 xs == product xs
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

2.12. Conjunción sobre una lista

Ejercicio 2.12. *Redefinir la función* `and` *tal que* `and l` *se verifica si todos los elementos de* `l` *son verdaderos. Por ejemplo,*

```
and [1<2, 2<3, 1 /= 0] ⇝ True
and [1<2, 2<3, 1 == 0] ⇝ False
```

Solución: Presentamos distintas definiciones:

1. Definición recursiva:

```
n_and_1 :: [Bool] -> Bool
n_and_1 []     = True
n_and_1 (x:xs) = x && n_and_1 xs
```

2. Definición con plegado:

```
n_and_2 :: [Bool] -> Bool
n_and_2 = foldr (&&) True
```

3. Las definiciones son equivalentes:

```
prop_equivalencia :: [Bool] -> Bool
prop_equivalencia xs =
    n_and_1 xs == and xs &&
    n_and_2 xs == and xs
```

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

2.13. Disyunción sobre una lista

Ejercicio 2.13. *Redefinir la función* `or` *tal que* `or l` *se verifica si algún elemento de* `l` *es verdadero. Por ejemplo,*

```
or [1<2, 2<3, 1 /= 0] ⇝ True
or [3<2, 4<3, 1 == 0] ⇝ False
```

Solución: Presentamos distintas definiciones:

1. Definición recursiva:

```
n_or_1 :: [Bool] -> Bool
n_or_1 []     = False
n_or_1 (x:xs) = x || n_or_1 xs
```

2. Definición con plegado:

```
n_or_2 :: [Bool] -> Bool
n_or_2 = foldr (||) False
```

3. Las definiciones son equivalentes:

```
prop_equivalencia :: [Bool] -> Bool
prop_equivalencia xs =
    n_or_1 xs == or xs &&
    n_or_2 xs == or xs
```

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

2.14. Plegado por la derecha

Ejercicio 2.14. *Redefinir la función* `foldr` *tal que* `foldr op e l` *pliega por la derecha la lista* `l` *colocando el operador* `op` *entre sus elementos y el elemento* `e` *al final. Es decir,*

```
foldr op e [x1,x2,x3]     ⇝ x1 op (x2 op (x3 op e))
foldr op e [x1,x2,...,xn] ⇝ x1 op (x2 op (... op (xn op e)))
```

Por ejemplo,

```
foldr (+) 3 [2,3,5] ⇝ 13
foldr (-) 3 [2,3,5] ⇝ 1
```

Solución:

```
n_foldr :: (a -> b -> b) -> b -> [a] -> b
n_foldr f e []     = e
n_foldr f e (x:xs) = f x (n_foldr f e xs)
```

2.15. Plegado por la izquierda

Ejercicio 2.15. *Redefinir la función* `foldl` *tal que* `foldl op e l` *pliega por la izquierda la lista* `l` *colocando el operador* `op` *entre sus elementos y el elemento* `e` *al principio. Es decir,*

```
foldl op e [x1,x2,x3]     ⇝ (((e op x1) op x2) op x3
foldl op e [x1,x2,...,xn] ⇝ (...((e op x1) op x2) ... op xn
```

Por ejemplo,

```
foldl (+) 3 [2,3,5] ⇝ 13
foldl (-) 3 [2,3,5] ⇝ -7
```

Solución: Definición recursiva

```
n_foldl :: (a -> b -> a) -> a -> [b] -> a
n_foldl f z []     = z
n_foldl f z (x:xs) = n_foldl f (f z x) xs
```

Las definiciones son equivalentes:

```
prop_equivalencia :: Int -> [Int] -> Bool
prop_equivalencia n xs =
    n_foldl (+) n xs == foldl (+) n xs
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

2.16. Resultados acumulados

Ejercicio 2.16. *Redefinir la función* `scanr` *tal que* `scanr op e l` *pliega por la derecha la lista* `l` *colocando el operador* `op` *entre sus elementos y el elemento* `e` *al final y escribe los resultados acumulados. Es decir,*

```
scanr op e [x1,x2,x3] ⇝ [x1 op (x2 op (x3 op e)),
                         x2 op (x3 op e),
                         x3 op e,
                         e]
```

Por ejemplo,

```
scanr (+) 3 [2,3,5] ⇝[13,11,8,3]
```

Solución:

```
n_scanr :: (a -> b -> b) -> b -> [a] -> [b]
n_scanr f q0 []     = [q0]
n_scanr f q0 (x:xs) = f x q : qs
                      where qs@(q:_) = n_scanr f q0 xs
```

Las definiciones son equivalentes:

```
prop_equivalencia :: Int -> [Int] -> Bool
prop_equivalencia n xs =
    n_scanr (+) n xs == scanr (+) n xs
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

2.17. Lista de factoriales

Ejercicio 2.17. *Definir la función* `factoriales` *tal que* `factoriales n` *es la lista de los factoriales desde el factorial de* `0` *hasta el factorial de* `n`*. Por ejemplo,*

```
factoriales 5 ⇝ [1,1,2,6,24,120]
```

Solución: Se presentan distintas definiciones:

1. Definición recursiva

```
factoriales_1 :: Integer -> [Integer]
factoriales_1 n =
    reverse (aux n)
    where aux 0     = [1]
          aux (n+1) = (factorial (n+1)) : aux n
```

2. Definición recursiva con acumuladores:

```
factoriales_2 :: Integer -> [Integer]
factoriales_2 n =
    reverse (aux (n+1) 0 [1])
    where aux n m (x:xs) = if n==m then xs
                                   else aux n (m+1) (((m+1)*x):x:xs)
```

3. Definición con listas intensionales:

```
factoriales_3 :: Integer -> [Integer]
factoriales_3 n = [factorial x | x <- [0..n]]
```

4. Definición con `map`:

```
factoriales_4 :: Integer -> [Integer]
factoriales_4 n = map factorial [0..n]
```

5. Definición con `scanl`:

```
factoriales_5 :: Integer -> [Integer]
factoriales_5 n = scanl (*) 1 [1..n]
```

Las definiciones son equivalentes:

```
prop_equivalencia :: Integer -> Property
prop_equivalencia n =
    n >= 0 ==>
    (factoriales_5 n == factoriales_1 n &&
     factoriales_2 n == factoriales_1 n &&
     factoriales_3 n == factoriales_1 n &&
     factoriales_4 n == factoriales_1 n &&
     factoriales_5 n == factoriales_1 n)
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

Se puede observar la eficiencia relativa en la siguiente sesión

```
Main> :set +s
Main> factoriales_1 100
[1,1,2,6,24,120,720,5040,40320,362880,3628800,39916800,...]
(171696 reductions, 322659 cells)
Main> factoriales_2 100
[1,1,2,6,24,120,720,5040,40320,362880,3628800,39916800,...]
(2457 reductions, 13581 cells)
Main> factoriales_3 100
[1,1,2,6,24,120,720,5040,40320,362880,3628800,39916800,...]
(169929 reductions, 319609 cells)
Main> factoriales_4 100
[1,1,2,6,24,120,720,5040,40320,362880,3628800,39916800,...]
(169930 reductions, 319611 cells)
Main> factoriales_5 100
[1,1,2,6,24,120,720,5040,40320,362880,3628800,39916800,...]
(2559 reductions, 12876 cells)
Main>
```

Se observa que las más eficientes son la 2 y la 5.

2.18. Iteración hasta–que

Ejercicio 2.18. *Redefinir la función* `until` *tal que* `until p f x` *aplica la* `f` *a* `x` *el menor número posible de veces, hasta alcanzar un valor que satisface el predicado* `p`*. Por ejemplo,*

```
until (>1000) (2*) 1 ⇝ 1024
```

Solución:

```
n_until :: (a -> Bool) -> (a -> a) -> a -> a
n_until p f x  = if p x then x else n_until p f (f x)
```

2.19. Composición de funciones

Ejercicio 2.19. *Redefinir la función* `(.)` *tal que* `f . g` *es la composición de las funciones* `f` *y* `g`*; es decir, la función que aplica* `x` *en* `f(g(x))`*. Por ejemplo,*

```
(cuadrado . siguiente) 2 ⇝ 9
(siguiente . cuadrado) 2 ⇝ 5
```

Solución:

```
compuesta :: (b -> c) -> (a -> b) -> (a -> c)
(f `compuesta` g) x  = f (g x)
```

Por ejemplo,

```
(cuadrado `compuesta` siguiente) 2 ⇝ 9
(siguiente `compuesta` cuadrado) 2 ⇝ 5
```

2.20. Intercambio de orden de argumentos

Ejercicio 2.20. *Redefinir la función* `flip` *que intercambia el orden de sus argumentos. Por ejemplo,*

```
flip (-) 5 2 ⇝ -3
flip (/) 5 2 ⇝ 0.4
```

Solución:

```
flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x
```

2.21. Relación de divisibilidad

Ejercicio 2.21. *Definir la función* `divisible` *tal que* `divisible x y` *se verifica si* `x` *es divisible por* `y`*. Por ejemplo,*

```
divisible 9 3 ⇝ True
divisible 9 2 ⇝ False
```

Solución:

```
divisible :: Int -> Int -> Bool
divisible x y  = x `rem` y == 0
```

2.22. Lista de divisores de un número

Ejercicio 2.22. *Definir la función* `divisores` *tal que* `divisores x` *es la lista de los divisores de* `x`*. Por ejemplo,*

```
divisores 12 ⇝ [1,2,3,4,6,12]
```

Solución: Se presentan distintas definiciones:

1. Mediante filtro:

```
divisores_1 :: Int -> [Int]
divisores_1 x  = filter (divisible x) [1..x]
```

2. Mediante comprensión:

```
divisores_2 :: Int -> [Int]
divisores_2 x  = [y | y <- [1..x], divisible x y]
```

3. Equivalencia de las definiciones:

```
prop_equivalencia_1_2 x =
    divisores_1 x == divisores_2 x
```

   Compobación:

```
Divisores> quickCheck prop_equivalencia_1_2
OK, passed 100 tests.
```

4. Usaremos como `divisores` la segunda

```
divisores = divisores_2
```

2.23. Comprobación de número primo

Ejercicio 2.23. *Definir la función* `primo` *tal que* `primo x` *se verifica si* `x` *es primo. Por ejemplo,*

```
primo 5 ⇝ True
primo 6 ⇝ False
```

Solución:

```
primo :: Int -> Bool
primo x  = divisores x == [1,x]
```

2.24. Lista de primos

Ejercicio 2.24. *Definir la función* `primos` *tal que* `primos x` *es la lista de los números primos menores o iguales que* `x`*. Por ejemplo,*

```
primos 40 ⇝ [2,3,5,7,11,13,17,19,23,29,31,37]
```

Solución: Se presentan distintas definiciones:

1. Mediante filtrado:

```
primos_1 :: Int -> [Int]
primos_1 x  = filter primo [1..x]
```

2. Mediante comprensión:

```
primos_2 :: Int -> [Int]
primos_2 x = [y | y <- [1..x], primo y]
```

2.25. Cálculo del día de la semana

Ejercicio 2.25. *Definir la función* `día` *tal que* `dia d m a` *es el día de la semana correspondiente al día* `d` *del mes* `m` *del año* `a`*. Por ejemplo,*

```
día 31 12 2007 ⇝ "lunes"
```

Solución:

```
día d m a = díaSemana ((númeroDeDías d m a) `mod` 7)
```

donde se usan las siguientes funciones auxiliares

- `númeroDía d m a` es el número de días trancurridos desde el 1 de enero del año 0 hasta el día `d` del mes `m` del año `a`. Por ejemplo,

```
númeroDeDías 31 12 2007 ⇝ 733041
```

```
númeroDeDías d m a =   (a-1)*365
                     + númeroDeBisiestos a
                     + sum (take (m-1) (meses a))
                     + d
```

- `númeroDeBisiestos a` es el número de años bisiestos antes del año `a`.

```
númeroDeBisiestos a = length (filter bisiesto [1..a-1])
```

- `bisiesto a` se verifica si el año `a` es bisiesto. La definición de año bisiesto es
  - un año divisible por 4 es un año bisiesto (por ejemplo 1972);
  - excepción: si es divisible por 100, entonces no es un año bisiesto
  - excepción de la excepción: si es divisible por 400, entonces es un año bisiesto (por ejemplo 2000).

```
bisiesto a =
    divisible a 4 && (not(divisible a 100) || divisible a 400)
```

- `meses a` es la lista con el número de días del los meses del año `a`. Por ejemplo,

```
meses 2000 ⇝ [31,29,31,30,31,30,31,31,30,31,30,31]
```

```
meses a = [31, feb, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
    where feb | bisiesto a = 29
              | otherwise  = 28
```

- `díaSemana n` es el `n`–ésimo día de la semana comenzando con 0 el domingo. Por ejemplo,

```
díaSemana 2 ⇝ "martes"
```

```
díaSemana 0 = "domingo"
díaSemana 1 = "lunes"
díaSemana 2 = "martes"
díaSemana 3 = "miércoles"
díaSemana 4 = "jueves"
díaSemana 5 = "viernes"
díaSemana 6 = "sábado"
```

2.26. Diferenciación numérica

Ejercicio 2.26. *Definir la función* `derivada` *tal que* `derivada a f x` *es el valor de la derivada de la función* `f` *en el punto* `x` *con aproximación* `a`*. Por ejemplo,*

```
derivada 0.001 sin pi ⇝ -0.9999273
derivada 0.001 cos pi ⇝ 0.0004768371
```

Solución:

```
derivada :: Float -> (Float -> Float) -> Float -> Float
derivada a f x  = (f(x+a)-f(x))/a
```

Ejercicio 2.27. *Definir las siguientes versiones de* `derivada`*:*

- `derivadaBurda` *cuando la aproximación es 0.01.*
- `derivadaFina` *cuando la aproximación es 0.0001.*
- `derivadaSuper` *cuando la aproximación es 0.000001.*

Por ejemplo,

```
derivadaFina cos pi ⇝ 0.0
derivadaFina sin pi ⇝ -0.9989738
```

Solución:

```
derivadaBurda = derivada 0.01
derivadaFina  = derivada 0.0001
derivadaSuper = derivada 0.000001
```

Ejercicio 2.28. *Definir la función* `derivadaFinaDelSeno` *tal que* `derivadaFinaDelSeno x` *es el valor de la derivada fina del seno en* `x`*. Por ejemplo,*

```
derivadaFinaDelSeno pi ⇝ -0.9989738
```

Solución:

```
derivadaFinaDelSeno = derivadaFina sin
```

2.27. Cálculo de la raíz cuadrada

Ejercicio 2.29. *Definir la función* `RaizCuadrada` *tal que* `raiz x` *es la raíz cuadrada de* `x` *calculada usando la siguiente propiedad*

Si y es una aproximación de $\sqrt{x}$, entonces $\frac{1}{2}(y + \frac{x}{y})$ es una aproximación mejor.

Por ejemplo,

```
raizCuadrada 9 ⇝ 3.00000000139698
```

Solución:

```
raizCuadrada :: Double -> Double
raizCuadrada x = until aceptable mejorar 1
    where mejorar y   = 0.5*(y+x/y)
          aceptable y = abs(y*y-x) < 0.00001
```

2.28. Cálculo de ceros de una función

Ejercicio 2.30. *Definir la función* `puntoCero` *tal que* `puntoCero f` *es un cero de la función* `f` *calculado usando la siguiente propiedad*

> *Si b es una aproximación para el punto cero de f, entonces* $b - \frac{f(b)}{f'(b)}$ *es una mejor aproximación.*

Por ejemplo,

```
puntoCero cos ⇝ 1.570796
```

Solución:

```
puntoCero f = until aceptable mejorar 1
    where mejorar b   = b - f b / derivadaFina f b
          aceptable b = abs (f b) < 0.00001
```

Ejercicio 2.31. *Usando* `puntoCero`*, definir las siguientes funciones:*

- `raíz_cuadrada` *tal que* `raíz_cuadrada x` *es la raíz cuadrada de* `x`*.*
- `raíz_cúbica` *tal que* `raíz_cúbica x` *es la raíz cúbica de* `x`*.*
- `arco_seno` *tal que* `arco_seno x` *es el arco cuyo seno es* `x`*.*
- `arco_coseno` *tal que* `arco_coseno x` *es el arco cuyo coseno es* `x`*.*

Solución:

```
raíz_cuadrada_1 a = puntoCero f
    where f x = x*x-a

raíz_cúbica_1 a = puntoCero f
    where f x = x*x*x-a
```

```
arco_seno_1 a = puntoCero f
    where f x = sin x - a

arco_coseno_1 a = puntoCero f
    where f x = cos x - a
```

Ejercicio 2.32. *Usando* `puntoCero`*, definir la función* `inversa` *tal que* `inversa f` *es la inversa de la función* `f`*.*

Solución:

```
inversa g a = puntoCero f
    where f x = g x - a
```

Ejercicio 2.33. *Usando la función* `inversa`*, redefinir las funciones* `raíz_cuadrada`*,* `raíz_cúbica`*,* `arco_seno` *y* `arco_coseno`*.*

Solución:

```
raíz_cuadrada_2 = inversa (^2)
raíz_cúbica_2   = inversa (^3)
arco_seno_2     = inversa sin
arco_coseno_2   = inversa cos
```

Capítulo 3

Estructuras de datos

Contenido

3.1. Relación de igualdad entre listas

Ejercicio 3.1. *Definir la función* `igualLista` *tal que* `igualLista xs ys` *se verifica si las dos listas* `xs` *e* `ys` *son iguales. Por ejemplo,*

```
igualLista :: Eq a => [a] -> [a] -> Bool
igualLista [1,2,3,4,5] [1..5] ⇝ True
igualLista [1,3,2,4,5] [1..5] ⇝ False
```

Nota: `igualLista` *es equivalente a* `==`*.*

Solución:

```
igualLista :: Eq a => [a] -> [a] -> Bool
igualLista []     []     =  True
igualLista (x:xs) (y:ys) =  x==y && igualLista xs ys
igualLista _      _      =  False
```

Las definiciones son equivalentes:

```
prop_equivalencia :: [Int] -> [Int] -> Bool
prop_equivalencia xs ys =
    igualLista xs ys == (xs==ys)
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

3.2. Concatenación de listas

Ejercicio 3.2. *Definir la función* `conc` *tal que* `conc l1 l2` *es la concatenación de* `l1` *y* `l2`*. Por ejemplo,*

```
conc [2,3] [3,2,4,1] ⇝ [2,3,3,2,4,1]
```

Nota: `conc` *es equivalente a* `(++)`*.*

Solución:

```
conc :: [a] -> [a] -> [a]
conc []     ys  = ys
conc (x:xs) ys  = x : (conc xs ys)
```

Las definiciones son equivalentes:

```
prop_equivalencia :: [Int] -> [Int] -> Bool
prop_equivalencia xs ys =
    conc xs ys == xs++ys
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

3.3. Concatenación de una lista de listas

Ejercicio 3.3. *Redefinir la función* `concat` *tal que* `concat l` *es la concatenación de las lista de* `l`*. Por ejemplo,*

```
concat [[1,2,3],[4,5],[],[1,2]] ⇝ [1,2,3,4,5,1,2]
```

Solución: Presentamos distintas definiciones:

1. Definición recursiva:

   ```
   concat_1 :: [[a]] -> [a]
   concat_1 []       = []
   concat_1 (xs:xss) = xs ++ concat_1 xss
   ```

2. Definión con plegados:

   ```
   concat_2 :: [[a]] -> [a]
   concat_2 = foldr (++) []
   ```

3. Definión por comprensión:

   ```
   concat_3 :: [[a]] -> [a]
   concat_3 xss = [x | xs <- xss, x <- xs]
   ```

Las definiciones son equivalentes:

```
prop_equivalencia :: [[Int]] -> Bool
prop_equivalencia x =
    concat_1 x == concat x &&
    concat_2 x == concat x &&
    concat_3 x == concat x
```

Comprobación:

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

3.4. Cabeza de una lista

Ejercicio 3.4. *Redefinir la función* `head` *tal que* `head l` *es la cabeza de la lista* `l`*. Por ejemplo,*

```
head [3,5,2] ⇝ 3
head []      ⇝ Program error: pattern match failure: head []
```

Solución:

```
n_head :: [a] -> a
n_head (x:_) = x
```

3.5. Resto de una lista

Ejercicio 3.5. *Redefinir la función* `tail` *tal que* `tail l` *es el resto de la lista* `l`*. Por ejemplo,*

```
tail [3,5,2]    ⇝ [5,2]
tail (tail [1]) ⇝ Program error: pattern match failure: tail []
```

Solución:

```
n_tail :: [a] -> [a]
n_tail (_:xs) = xs
```

3.6. Último elemento

Ejercicio 3.6. *Redefinir la función* `last` *tal que* `last l` *es el último elemento de la lista* `l`*. Por ejemplo,*

```
last [1,2,3] ⇝ 3
last []      ⇝ Program error: pattern match failure: last []
```

Solución: Presentamos distintas definiciones:

1. Definición recursiva:

```
n_last_1 :: [a] -> a
n_last_1 [x]    = x
n_last_1 (_:xs) = n_last_1 xs
```

2. Con plegados:

```
n_last_2 :: [a] -> a
n_last_2 = foldr1 (\x y -> y)
```

3. Con `head` y `reverse`:

```
n_last_3 :: [a] -> a
n_last_3 xs = head (reverse xs)
```

4. Con `head`, `reverse` y `(.)`:

```
n_last_4 :: [a] -> a
n_last_4 = head . reverse
```

5. Con `(!!)` y `length`

```
n_last_5 :: [a] -> a
n_last_5 xs = xs !! (length xs - 1)
```

Las definiciones son equivalentes para las listas no vacía:

```
prop_equivalencia :: [Int] -> Property
prop_equivalencia xs =
    not (null xs) ==>
    (n_last_1 xs == last xs &&
     n_last_2 xs == last xs &&
     n_last_3 xs == last xs &&
     n_last_4 xs == last xs &&
     n_last_5 xs == last xs)
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

3.7. Lista sin el último elemento

Ejercicio 3.7. *Redefinir la función* `init` *tal que* `init l` *es la lista* `l` *sin el último elemento. Por ejemplo,*

```
init [1,2,3] ⇝ [1,2]
init [4]     ⇝ []
```

Solución: Presentamos distintas definiciones:

1. Definición recursiva:

```
n_init_1 :: [a] -> [a]
n_init_1 [x]    = []
n_init_1 (x:xs) = x : n_init_1 xs
```

2. Definición con `tail` y `reverse`:

```
n_init_2 :: [a] -> [a]
n_init_2 xs = reverse (tail (reverse xs))
```

3. Definición con `tail`, `reverse` y `(.)`:

```
n_init_3 :: [a] -> [a]
n_init_3 = reverse . tail . reverse
```

Las definiciones son equivalentes sobre listas no vacía:

```
prop_equivalencia :: [Int] -> Property
prop_equivalencia xs =
    not (null xs) ==>
    (n_init_1 xs == init xs &&
     n_init_2 xs == init xs &&
     n_init_3 xs == init xs)
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

3.8. Segmento inicial

Ejercicio 3.8. *Definir la función* `take` *tal que* `take n l` *es la lista de los* `n` *primeros elementos de* `l`*. Por ejemplo,*

```
take 2 [3,5,4,7]  ⇝ [3,5]
take 12 [3,5,4,7] ⇝ [3,5,4,7]
```

Solución:

```
n_take :: Int -> [a] -> [a]
n_take n _  | n <= 0  = []
n_take _ []           = []
n_take n (x:xs)       = x : n_take (n-1) xs
```

Las definiciones son equivalentes:

```
prop_equivalencia :: Int -> [Int] -> Bool
prop_equivalencia n xs =
    n_take n xs == take n xs
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

3.9. Segmento inicial filtrado

Ejercicio 3.9. *Redefinir la función* `takeWhile` *tal que* `takeWhile p l` *es la lista de los elementos iniciales de* `l` *que verifican el predicado* `p`*. Por ejemplo,*

```
takeWhile even [2,4,6,7,8,9] ⇝ [2,4,6]
```

Solución: Definición recursiva:

```
n_takeWhile           :: (a -> Bool) -> [a] -> [a]
n_takeWhile p []       = []
n_takeWhile p (x:xs)
         | p x       = x : n_takeWhile p xs
         | otherwise = []
```

Las definiciones son equivalentes:

```
prop_equivalencia :: [Int] -> Bool
prop_equivalencia xs =
    n_takeWhile even xs == takeWhile even xs
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

3.10. Segmento final

Ejercicio 3.10. *Redefinir la función* `drop` *tal que* `drop n l` *es la lista obtenida eliminando los primeros* `n` *elementos de la lista* `l`*. Por ejemplo,*

```
drop 2 [3..10] ⇝ [5,6,7,8,9,10]
drop 12 [3..10] ⇝ []
```

Solución:

```
n_drop :: Int -> [a] -> [a]
n_drop n xs | n <= 0 = xs
n_drop _ []          = []
n_drop n (_:xs)      = n_drop (n-1) xs
```

Las definiciones son equivalentes:

```
prop_equivalencia :: Int -> [Int] -> Bool
prop_equivalencia n xs =
    n_drop n xs == drop n xs
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

La relación entre los segmentos iniciales y finales es

```
prop_take_y_drop :: Int -> [Int] -> Bool
prop_take_y_drop n xs =
    n_take n xs ++ n_drop n xs == xs
```

Comprobación

```
Main> quickCheck prop_take_y_drop
OK, passed 100 tests.
```

3.11. Segmento final filtrado

Ejercicio 3.11. *Redefinir la función* `dropWhile` *tal que* `dropWhile p l` *es la lista* `l` *sin los elementos iniciales que verifican el predicado* `p`*. Por ejemplo,*

```
dropWhile even [2,4,6,7,8,9] ⇝ [7,8,9]
```

Solución:

```
n_dropWhile :: (a -> Bool) -> [a] -> [a]
n_dropWhile p []       = []
n_dropWhile p l@(x:xs)
         | p x       = n_dropWhile p xs
         | otherwise = l
```

Las definiciones son equivalentes:

```
prop_equivalencia :: [Int] -> Bool
prop_equivalencia xs =
    n_dropWhile even xs == dropWhile even xs
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

3.12. N–ésimo elemento de una lista

Ejercicio 3.12. *Definir la función* `nth` *tal que* `nth l n` *es elemento* `n`*–ésimo de* `l`*, empezando a numerar con el 0. Por ejemplo,*

```
nth [1,3,2,4,9,7] 3 ⇝ 4
```

Nota: `nth` *es equivalente a* `(!!)`*.*

Solución:

```
nth :: [a] -> Int -> a
nth (x:_)  0 = x
nth (_:xs) n = nth xs (n-1)
```

Las definiciones son equivalentes:

```
prop_equivalencia :: [Int] -> Int -> Property
prop_equivalencia xs n =
    0 < n && n < length xs ==> nth xs n == xs!!n
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

3.13. Inversa de una lista

Ejercicio 3.13. *Redefinir la función* `reverse` *tal que* `reverse l` *es la inversa de* `l`*. Por ejemplo,*

```
reverse [1,4,2,5] ⇝ [5,2,4,1]
```

Solución: Se presentan distintas definiciones:

1. Definición recursiva:

```
n_reverse_1 :: [a] -> [a]
n_reverse_1 []     = []
n_reverse_1 (x:xs) = n_reverse_1 xs ++ [x]
```

2. Definición recursiva con acumulador:

```
n_reverse_2 :: [a] -> [a]
n_reverse_2 xs =
    n_reverse_2_aux xs []
    where n_reverse_2_aux [] ys     = ys
          n_reverse_2_aux (x:xs) ys = n_reverse_2_aux xs (x:ys)
```

3. Con plegado:

```
n_reverse_3 :: [a] -> [a]
n_reverse_3 = foldl (flip (:)) []
```

Las definiciones son equivalentes:

```
prop_equivalencia :: [Int] -> Bool
prop_equivalencia xs =
    n_reverse_1 xs == reverse xs &&
    n_reverse_2 xs == reverse xs &&
    n_reverse_3 xs == reverse xs
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

Comparación de eficiencia : Número de reducciones al invertir la lista `[1..n]`

n	Def. 1	Def. 2	Def. 3
100	7.396	2.347	2.446
200	24.746	4.647	4.846
400	89.446	9.247	9.646
1000	523.546	23.047	24.046

3.14. Longitud de una lista

Ejercicio 3.14. *Redefinir la función* `length` *tal que* `length l` *es el número de elementos de* `l`. *Por ejemplo,*

```
length [1,3,6] ⇝ 3
```

Solución: Se presentan distintas definiciones:

1. Definición recursiva:

```
n_length_1 :: [a] -> Int
n_length_1 []     = 0
n_length_1 (_:xs) = 1 + n_length_1 xs
```

2. Definición con plegado por la derecha:

```
n_length_2 :: [a] -> Int
n_length_2 = foldr (\x y -> y+1) 0
```

3. Definición con plegado por la izquierda:

```
n_length_3 :: [a] -> Int
n_length_3 = foldl (\x y -> x+1) 0
```

4. Definición con `sum` y listas intensionales:

```
n_length_4 :: [a] -> Int
n_length_4 xs = sum [1 | x <- xs]
```

Las definiciones son equivalentes:

```
prop_equivalencia :: [Int] -> Bool
prop_equivalencia xs =
    n_length_1 xs == length xs &&
    n_length_2 xs == length xs &&
    n_length_3 xs == length xs &&
    n_length_4 xs == length xs
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

3.15. Comprobación de pertenencia de un elemento a una lista

Ejercicio 3.15. *Redefinir la función* `elem` *tal que* `elem e l` *se verifica si* `e` *es un elemento de* `l`. *Por ejemplo,*

```
elem 2 [1,2,3] ⇝ True
elem 4 [1,2,3] ⇝ False
```

Solución: Se presentan distintas definiciones:

1. Definición recursiva

```
n_elem_1 :: Eq a => a -> [a] -> Bool
n_elem_1 _ []     = False
n_elem_1 x (y:ys) = (x==y) || n_elem_1 x ys
```

2. Definición con plegado:

```
n_elem_2 :: Eq a => a -> [a] -> Bool
n_elem_2 x = foldl (\z y -> z || x==y) False
```

3. Definición con `or` y `map`

```
n_elem_3 :: Eq a => a -> [a] -> Bool
n_elem_3 x ys = or (map (==x) ys)
```

4. Definición con `or`, `map` y `(.)`

```
n_elem_4 :: Eq a => a -> [a] -> Bool
n_elem_4 x = or . map (==x)
```

5. Definición con `or` y lista intensional:

```
n_elem_5 :: Eq a => a -> [a] -> Bool
n_elem_5 x ys = or [x==y | y <- ys]
```

6. Definición con `any` y `(.)`

```
n_elem_6 :: Eq a => a -> [a] -> Bool
n_elem_6 = any . (==)
```

7. Definición con `not` y `notElem`

```
n_elem_7 :: Eq a => a -> [a] -> Bool
n_elem_7 x = not . (notElem x)
```

Las definiciones son equivalentes:

```
prop_equivalencia :: Int -> [Int] -> Bool
prop_equivalencia x ys =
    n_elem_1 x ys == elem x ys &&
    n_elem_2 x ys == elem x ys &&
    n_elem_3 x ys == elem x ys &&
    n_elem_4 x ys == elem x ys &&
    n_elem_5 x ys == elem x ys &&
    n_elem_6 x ys == elem x ys &&
    n_elem_7 x ys == elem x ys
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

3.16. Comprobación de no pertenencia de un elemento a una lista

Ejercicio 3.16. *Redefinir la función* `notElem` *tal que* `notElem e l` *se verifica si* `e` *no es un elemento de* `l`*. Por ejemplo,*

```
notElem 2 [1,2,3] ⇝ False
notElem 4 [1,2,3] ⇝ True
```

Solución: Se presentan distintas definiciones:

1. Definición recursiva

```
n_notElem_1 :: Eq a => a -> [a] -> Bool
n_notElem_1 _ []     = True
n_notElem_1 x (y:ys) = (x/=y) && n_notElem_1 x ys
```

2. Definición con plegado:

```
n_notElem_2 :: Eq a => a -> [a] -> Bool
n_notElem_2 x = foldl (\z y -> z && x/=y) True
```

3. Definición con `or` y `map`

```
n_notElem_3 :: Eq a => a -> [a] -> Bool
n_notElem_3 x ys = and (map (/=x) ys)
```

4. Definición con `or`, `map` y `(.)`

```
n_notElem_4 :: Eq a => a -> [a] -> Bool
n_notElem_4 x = and . map (/=x)
```

5. Definición con `or` y lista intensional:

```
n_notElem_5 :: Eq a => a -> [a] -> Bool
n_notElem_5 x ys = and [x/=y | y <- ys]
```

6. Definición con `any` y `(.)`

```
n_notElem_6 :: Eq a => a -> [a] -> Bool
n_notElem_6 = all . (/=)
```

7. Definición con `not` y `elem`

```
n_notElem_7 :: Eq a => a -> [a] -> Bool
n_notElem_7 x = not . (elem x)
```

Las definiciones son equivalentes:

```
prop_equivalencia :: Int -> [Int] -> Bool
prop_equivalencia x ys =
    n_notElem_1 x ys == notElem x ys &&
    n_notElem_2 x ys == notElem x ys &&
    n_notElem_3 x ys == notElem x ys &&
    n_notElem_4 x ys == notElem x ys &&
    n_notElem_5 x ys == notElem x ys &&
    n_notElem_6 x ys == notElem x ys &&
    n_notElem_7 x ys == notElem x ys
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

3.17. Comprobación de que una lista está ordenada

Ejercicio 3.17. *Definir la función* `lista_ordenada` *tal que* `lista_ordenada l` *se verifica si la lista* `l` *está ordenada de menor a mayor. Por ejemplo,*

```
lista_ordenada [1,3,3,5] ⇝ True
lista_ordenada [1,3,5,3] ⇝ False
```

Solución: Se presentan distintas definiciones:

1. Definición recursiva

```
lista_ordenada_1 :: Ord a => [a] -> Bool
lista_ordenada_1 []       = True
lista_ordenada_1 [_]      = True
lista_ordenada_1 (x:y:xs) = (x <= y) && lista_ordenada_1 (y:xs)
```

2. Definición con `and`, y `zipWith`

```
lista_ordenada_2 :: Ord a => [a] -> Bool
lista_ordenada_2 []       = True
lista_ordenada_2 [_]      = True
lista_ordenada_2 xs       = and (zipWith (<=) xs (tail xs))
```

3. Usaremos como `lista_ordenada` la primera

```
lista_ordenada :: Ord a => [a] -> Bool
lista_ordenada = lista_ordenada_1
```

Las definiciones son equivalentes:

```
prop_equivalencia :: [Int] -> Bool
prop_equivalencia xs =
    lista_ordenada_1 xs == lista_ordenada_2 xs
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

3.18. Comprobación de la igualdad de conjuntos

Ejercicio 3.18. *Definir la función* `igual_conjunto` *tal que* `igual_conjunto l1 l2` *se verifica si las listas* `l1` *y* `l2` *vistas como conjuntos son iguales Por ejemplo,*

```
igual_conjunto [1..10] [10,9..1]  ⇝ True
igual_conjunto [1..10] [11,10..1] ⇝ False
```

Solución: Se presentan distintas definiciones:

1. Usando `subconjunto`

```
igual_conjunto_1 :: Eq a => [a] -> [a] -> Bool
igual_conjunto_1 xs ys = subconjunto xs ys && subconjunto ys xs
```

2. Por recursión.

```
igual_conjunto_2 :: Eq a => [a] -> [a] -> Bool
igual_conjunto_2 xs ys = aux (nub xs) (nub ys)
    where aux []    []    = True
          aux (x:_) []    = False
          aux []    (y:_) = False
          aux (x:xs) ys   = x `elem` ys && aux xs (delete x ys)
```

3. Usando `sort`

```
igual_conjunto_3 :: (Eq a, Ord a) => [a] -> [a] -> Bool
igual_conjunto_3 xs ys = sort (nub xs) == sort (nub ys)
```

4. Usaremos como `igual_conjunto` la primera

```
igual_conjunto :: Eq a => [a] -> [a] -> Bool
igual_conjunto = igual_conjunto_1
```

Las definiciones son equivalentes:

```
prop_equivalencia :: [Int] -> [Int] -> Bool
prop_equivalencia xs ys =
    igual_conjunto_2 xs ys == igual_conjunto_1 xs ys &&
    igual_conjunto_3 xs ys == igual_conjunto_1 xs ys
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

3.19. Inserción ordenada de un elemento en una lista

Ejercicio 3.19. *Definir la función* `inserta` *tal que* `inserta e l` *inserta el elemento* `e` *en la lista* `l` *delante del primer elemento de* `l` *mayor o igual que* `e`*. Por ejemplo,*

```
inserta 5 [2,4,7,3,6,8,10] ⇝ [2,4,5,7,3,6,8,10]
```

Solución:

```
inserta :: Ord a => a -> [a] -> [a]
inserta e []     = [e]
inserta e (x:xs)
    | e<=x       = e:x:xs
    | otherwise  = x : inserta e xs
```

Se puede definir el generador de QuickCheck para que cree lista ordenadas y compruebe que `inserta` las mantiene ordenada

```
listaOrdenada :: Gen [Integer]
listaOrdenada =
  do n <- arbitrary
     listaDesde n
  where listaDesde n =
              frequency
               [(1, return []),
```

```
                 (5, do i <- arbitrary
                        ns <- listaDesde (n+abs i)
                        return (n:ns))]

prop_inserta x =
    forAll listaOrdenada $ \xs ->
    collect (length xs) $
    lista_ordenada (inserta x xs)
```

En efecto,

```
Inserta> quickCheck prop_inserta
OK, passed 100 tests.
18% 2.
16% 1.
10% 5.
10% 0.
9% 6.
7% 7.
7% 3.
4% 9.
4% 12.
3% 4.
2% 8.
2% 19.
2% 13.
1% 22.
1% 18.
1% 17.
1% 15.
1% 11.
1% 10.
```

3.20. Ordenación por inserción

Ejercicio 3.20. *Definir la función* `ordena_por_inserción` *tal que* `ordena_por_inserción l` *es la lista* `l` *ordenada mediante inserción, Por ejemplo,*

```
ordena_por_inserción [2,4,3,6,3] ⇝ [2,3,3,4,6]
```

Solución: Se presentan distintas definiciones:

1. Definición recursiva

```
ordena_por_inserción_1 :: Ord a => [a] -> [a]
ordena_por_inserción_1 []     = []
ordena_por_inserción_1 (x:xs) = inserta x (ordena_por_inserción_1 xs)
```

2. Definición por plegado por la derecha

```
ordena_por_inserción_2 :: Ord a => [a] -> [a]
ordena_por_inserción_2 = foldr inserta []
```

3. Definición por plegado por la izquierda

```
ordena_por_inserción_3 :: Ord a => [a] -> [a]
ordena_por_inserción_3 = foldl (flip inserta) []
```

Las definiciones son equivalentes:

```
prop_equivalencia :: [Int] -> Bool
prop_equivalencia xs =
    ordena_por_inserción_2 xs == ordena_por_inserción_1 xs &&
    ordena_por_inserción_2 xs == ordena_por_inserción_1 xs
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

Al comparar la eficiencia

```
Main> :set +s
Main> ordena_por_inserción_1 [100,99..1]
,,,
(51959 reductions, 68132 cells)
Main> ordena_por_inserción_2 [100,99..1]
,,,
(51960 reductions, 68034 cells)
Main> ordena_por_inserción_3 [100,99..1]
...
(3451 reductions, 5172 cells)
```

se observa que la tercera definición es más eficiente.

En los sucesivo usaremos como `ordena_por_inserción` la tercera

```
ordena_por_inserción :: Ord a => [a] -> [a]
ordena_por_inserción = ordena_por_inserción_2
```

El valor de `ordena_por_inserción` es una lista ordenada

```
prop_ordena_por_inserción_ordenada :: [Int] -> Bool
prop_ordena_por_inserción_ordenada xs =
    lista_ordenada (ordena_por_inserción xs)
```

```
Ordena_por_insercion> quickCheck prop_ordena_por_inserción_ordenada
OK, passed 100 tests.
```

3.21. Mínimo elemento de una lista

Ejercicio 3.21. *Redefinir la función* `minimum` *tal que* `minimum l` *es el menor elemento de la lista* `l`*. Por ejemplo,*

```
minimum [3,2,5] ⇝ 2
```

Solución: Se presentan distintas definiciones:

1. Definición recursiva:

```
n_minimum_1 :: Ord a => [a] -> a
n_minimum_1 [x] = x
n_minimum_1 (x:y:xs) = n_minimum_1 ((min x y):xs)
```

2. Definición con plegado:

```
n_minimum_2 :: Ord a => [a] -> a
n_minimum_2 = foldl1 min
```

3. Definición mediante ordenación:

```
n_minimum_3 :: Ord a => [a] -> a
n_minimum_3 = head . ordena_por_inserción
```

Las definiciones son equivalentes:

```
prop_equivalencia :: [Int] -> Property
prop_equivalencia xs =
    not (null xs) ==>
    (n_minimum_1 xs == minimum xs &&
     n_minimum_2 xs == minimum xs &&
     n_minimum_3 xs == minimum xs )
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

La eficiencia de las tres definiciones es equivalente:

```
Main :set +s
Main> n_minimum_1 [100,99..1]
1
(2644 reductions, 3568 cells)
Main> n_minimum_2 [100,99..1]
1
(2548 reductions, 3373 cells)
Main> n_minimum_3 [100,99..1]
1
(2552 reductions, 3477 cells)
```

La complejidad de `minimum_3` es lineal:

```
minimum_3 [10,9..1]       (   300 reductions,    416 cells)
minimum_3 [100,99..1]     (  2550 reductions,   3476 cells)
minimum_3 [1000,999..1]   ( 25050 reductions,  34076 cells)
minimum_3 [10000,9999..1] (250050 reductions, 340077 cells)
```

aunque la complejidad de `ordena_por_inserción` es cuadrática

```
ordena_por_inserción [10,9..1]       (      750 reductions,      1028 cells)
ordena_por_inserción [100,99..1]     (    51960 reductions,     68034 cells)
ordena_por_inserción [1000,999..1]   (  5019060 reductions,   6530485 cells)
ordena_por_inserción [10000,9999..1] (500190060 reductions, 650313987 cells)
```

3.22. Mezcla de dos listas ordenadas

Ejercicio 3.22. *Definir la función* `mezcla` *tal que* `mezcla l1 l2` *es la lista ordenada obtenida al mezclar las listas ordenadas* `l1` *y* `l2`*. Por ejemplo,*

```
mezcla [1,3,5] [2,9] ⇝ [1,2,3,5,9]
```

Solución: Se presentan distintas definiciones:

1. Definición recursiva:

```
mezcla_1 :: Ord a => [a] -> [a] -> [a]
mezcla_1 [] ys         = ys
mezcla_1 xs []         = xs
mezcla_1 (x:xs) (y:ys)
    | x <= y         = x : mezcla_1 xs (y:ys)
    | otherwise      = y : mezcla_1 (x:xs) ys
```

2. Definición recursiva con `inserta`:

```
mezcla_2 :: Ord a => [a] -> [a] -> [a]
mezcla_2 [] ys     = ys
mezcla_2 (x:xs) ys =  inserta x (mezcla_2 xs ys)
```

3. Usaremos como `mezcla` la primera

```
mezcla :: Ord a => [a] -> [a] -> [a]
mezcla = mezcla_1
```

Las definiciones son equivalentes:

```
prop_equivalencia :: [Int] -> [Int] -> Bool
prop_equivalencia xs ys =
    mezcla_1 xs ys == mezcla_2 xs ys
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

Las mezcla de listas ordenadas es una lista ordenada

```
prop_mezcla_ordenada :: [Int] -> [Int] -> Property
prop_mezcla_ordenada xs ys =
    lista_ordenada xs && lista_ordenada ys ==>
    lista_ordenada (mezcla xs ys)
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

3.23. Ordenación por mezcla

Ejercicio 3.23. *Definir la función* `ordena_por_m` *tal que* `ordena_por_m l` *es la lista* `l` *ordenada mediante mezclas, Por ejemplo,*

```
ordena_por_m [2,4,3,6,3] ⇝ [2,3,3,4,6]
```

Solución: Definición recursiva

```
ordena_por_m :: Ord a => [a] -> [a]
ordena_por_m []  = []
ordena_por_m [x] = [x]
ordena_por_m xs  = mezcla (ordena_por_m ys) (ordena_por_m zs)
    where medio = (length xs) `div` 2
          ys    = take medio xs
          zs    = drop medio xs
```

El valor de `ordena_por_m` es una lista ordenada

```
prop_ordena_por_m_ordenada :: [Int] -> Bool
prop_ordena_por_m_ordenada xs =
    lista_ordenada (ordena_por_m xs)
```

```
PD> quickCheck prop_ordena_por_m_ordenada
OK, passed 100 tests.
```

3.24. Dígito correspondiente a un carácter numérico

Ejercicio 3.24. *Definir la función* `dígitoDeCarácter` *tal que* `dígitoDeCarácter c` *es el dígito correspondiente al carácter numérico* `c`*. Por ejemplo,*

```
dígitoDeCarácter '3' ⇝ 3
```

Solución:

```
dígitoDeCarácter :: Char -> Int
dígitoDeCarácter c = ord c - ord '0'
```

3.25. Carácter correspondiente a un dígito

Ejercicio 3.25. *Definir la función* `carácterDeDígito` *tal que* `carácterDeDígito n` *es el carácter correspondiente al dígito* `n`*. Por ejemplo,*

```
carácterDeDígito 3 ⇝ '3'
```

Solución:

```
carácterDeDígito :: Int -> Char
carácterDeDígito n  = chr (n + ord '0')
```

La función `carácterDeDígito` es inversa de `dígitoDeCarácter`

```
prop_inversa :: Bool
prop_inversa =
    and [dígitoDeCarácter(carácterDeDígito d)==d | d <- [0..9]]
```

Comprobación

```
prop_inversa ⇝ True
```

3.26. Lista infinita de números

Ejercicio 3.26. *Definir la función* `desde` *tal que* `desde n` *es la lista de los números enteros a partir de* `n`*. Por ejemplo,*

```
desde 5     ⇝ [5,6,7,8,9,10,11,12,13,14,{Interrupted!}
```

se interrumpe con `Control-C`*.*

Solución: Se presentan distintas definiciones:

1. Definición recursiva:

```
desde_1 :: Int -> [Int]
desde_1 n = n : desde_1 (n+1)
```

2. Definición con segmento numérico:

```
desde_2 :: Int -> [Int]
desde_2 n = [n..]
```

Las definiciones son equivalentes:

```
prop_equivalencia :: Int -> Int -> Bool
prop_equivalencia n m =
    take m (desde_1 n) == take m (desde_2 n)
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

3.27. Lista con un elemento repetido

Ejercicio 3.27. *Redefinir la función* `repeat` *tal que* `repeat x` *es una lista infinita con el único elemento* `x`*. Por ejemplo,*

```
repeat 'a' ⇝ "aaaaaaaaaaaaaaaaaaaaaaaaaaa{Interrupted!}
```

Solución: Se presentan distintas definiciones:

1. Definición recursiva:

```
n_repeat_1 :: a -> [a]
n_repeat_1 x = x : n_repeat_1 x
```

2. Definición recursiva con entorno local

```
n_repeat_2 :: a -> [a]
n_repeat_2 x = xs where xs = x:xs
```

3. Definición con lista de comprensión:

```
n_repeat_3 :: a -> [a]
n_repeat_3 x = [x | y <- [1..]]
```

Las definiciones son equivalentes:

```
prop_equivalencia :: Int -> Int -> Bool
prop_equivalencia n m =
    take n (n_repeat_1 m) == take n (repeat m) &&
    take n (n_repeat_2 m) == take n (repeat m) &&
    take n (n_repeat_3 m) == take n (repeat m)
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

3.28. Lista con un elemento repetido un número dado de veces

Ejercicio 3.28. *Redefinir la función* `replicate` *tal que* `replicate n x` *es una lista con* `n` *copias del elemento* `x`*. Por ejemplo,*

```
replicate 10 3    ⇝ [3,3,3,3,3,3,3,3,3,3]
replicate (-10) 3 ⇝ []
```

Solución: Se presentan distintas definiciones:

1. Definición recursiva:

```
n_replicate_1 :: Int -> a -> [a]
n_replicate_1 (n+1) x = x : n_replicate_1 n x
n_replicate_1 _     x = []
```

2. Definición por comprensión:

```
n_replicate_2 :: Int -> a -> [a]
n_replicate_2 n x = [x | y <- [1..n]]
```

3. Definición usando `take` y `repeat`:

```
n_replicate_3 :: Int -> a -> [a]
n_replicate_3 n x = take n (repeat x)
```

Las definiciones son equivalentes:

```
prop_equivalencia :: Int -> Int -> Bool
prop_equivalencia n m =
    n_replicate_1 n m == replicate n m &&
    n_replicate_2 n m == replicate n m &&
    n_replicate_3 n m == replicate n m
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

3.29. Iteración de una función

Ejercicio 3.29. *Redefinir la función* `iterate` *tal que* `iterate f x` *es la lista cuyo primer elemento es* `x` *y los siguientes elementos se calculan aplicando la función* `f` *al elemento anterior. Por ejemplo,*

```
iterate (+1) 3          ⇝ [3,4,5,6,7,8,9,10,11,12,{Interrupted!}
iterate (*2) 1          ⇝ [1,2,4,8,16,32,64,{Interrupted!}
iterate (`div` 10) 1972 ⇝ [1972,197,19,1,0,0,0,0,0,0,{Interrupted!}
```

Solución:

```
n_iterate :: (a -> a) -> a -> [a]
n_iterate f x  = x : n_iterate f (f x)
```

Las definiciones son equivalentes:

```
prop_equivalencia :: Int -> Int -> Bool
prop_equivalencia n m =
    take n (n_iterate (+1) m) == take n (iterate (+1) m)
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

3.30. Conversión de número entero a cadena

Ejercicio 3.30. *Definir la función* `deEnteroACadena` *tal que* `deEnteroACadena n` *es la cadena correspondiente al número entero* `n`*. Por ejemplo,*

```
deEnteroACadena 1972 ⇝ "1972"
```

Solución: Se presentan distintas definiciones:

1. Mediante composición de funciones:

```
deEnteroACadena_1 :: Int -> String
deEnteroACadena_1 = map carácterDeDígito
                    . reverse
                    . map (`rem` 10)
                    . takeWhile (/= 0)
                    . iterate (`div` 10)
```

Ejemplo de cálculo

```
iterate (‘div‘ 10) 1972                      ⇝ [1972,197,19,1,0,0,0,...
(takeWhile (/= 0) . iterate (‘div‘ 10)) 1972 ⇝ [1972,197,19,1]
map (‘rem‘ 10) [1972,197,19,1]               ⇝ [2,7,9,1]
reverse [2,7,9,1]                            ⇝ [1,9,7,2]
map carácterDeDígito [1,9,7,2]               ⇝ "1972"
```

2. Mediante la función `show`

```
deEnteroACadena_2 :: Int -> String
deEnteroACadena_2 = show
```

Las definiciones son equivalentes:

```
prop_equivalencia :: Int -> Property
prop_equivalencia n =
    n > 0 ==>
    deEnteroACadena_1 n == deEnteroACadena_2 n
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

3.31. Cálculo de primos mediante la criba de Erastótenes

Ejercicio 3.31. *Definir la función* `primos_por_criba` *tal que* `primos_por_criba` *es la lista de los números primos mediante la criba de Erastótenes.*

```
primos_por_criba         ⇝ [2,3,5,7,11,13,17,19,23,29,{Interrupted!}
take 10 primos_por_criba ⇝ [2,3,5,7,11,13,17,19,23,29]
```

Solución: Se presentan distintas definiciones:

1. Definición perezosa:

```
primos_por_criba_1 :: [Int]
primos_por_criba_1  = map head (iterate eliminar [2..])
    where eliminar (x:xs) = filter (no_multiplo x) xs
          no_multiplo x y = y ‘mod‘ x /= 0
```

Para ver el cálculo, consideramos la siguiente variación

```
primos_por_criba_1_aux  = map (take 10) (iterate eliminar [2..])
    where eliminar (x:xs) = filter (no_multiplo x) xs
          no_multiplo x y = y `mod` x /= 0
```

Entonces,

```
Main> take 5 primos_por_criba_1_aux
[[ 2, 3, 4, 5, 6, 7, 8, 9,10,11],
 [ 3, 5, 7, 9,11,13,15,17,19,21],
 [ 5, 7,11,13,17,19,23,25,29,31],
 [ 7,11,13,17,19,23,29,31,37,41],
 [11,13,17,19,23,29,31,37,41,43]]
```

2. Definición por comprensión:

```
primos_por_criba_2 :: [Int]
primos_por_criba_2 =
    criba [2..]
    where criba (p:xs) = p : criba [n | n<-xs, n `mod` p /= 0]
```

Las definiciones son equivalentes:

```
prop_equivalencia :: Int -> Bool
prop_equivalencia n =
    take n primos_por_criba_1 == take n primos_por_criba_2
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

3.32. Comprobación de que todos los elementos son pares

Ejercicio 3.32. *Definir la función* `todosPares` *tal que*

`todosPares xs` *se verifica si todos los elementos de la lista* `xs` *son pares. Por ejemplo,*

```
todosPares [2,4,6]   ⇝ True
todosPares [2,4,6,7] ⇝ False
```

Solución: Se presentan distintas definiciones:

1. Definición recursiva:

```
todosPares_1 :: [Int] -> Bool
todosPares_1 []     = True
todosPares_1 (x:xs) = even x && todosPares_1 xs
```

2. Definición con `all`:

```
todosPares_2 :: [Int] -> Bool
todosPares_2 = all even
```

3. Definición por comprensión:

```
todosPares_3 :: [Int] -> Bool
todosPares_3 xs = ([x | x<-xs, even x] == xs)
```

Las definiciones son equivalentes:

```
prop_equivalencia :: [Int] -> Bool
prop_equivalencia xs =
    todosPares_2 xs == todosPares_1 xs &&
    todosPares_3 xs == todosPares_1 xs
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

3.33. Comprobación de que todos los elementos son impares

Ejercicio 3.33. *Definir la función* `todosImpares` *tal que*

`todosImpares xs` *se verifica si todos los elementos de la lista* `xs` *son impares. Por ejemplo,*

```
todosImpares [1,3,5]   ⇝ True
todosImpares [1,3,5,6] ⇝ False
```

Solución: Se presentan distintas definiciones:

1. Definición recursiva:

```
todosImpares_1 :: [Int] -> Bool
todosImpares_1 []     = True
todosImpares_1 (x:xs) = odd x && todosImpares_1 xs
```

2. Definición con `all`:

```
todosImpares_2 :: [Int] -> Bool
todosImpares_2 = all odd
```

3. Definición por comprensión:

```
todosImpares_3 :: [Int] -> Bool
todosImpares_3 xs = ([x | x<-xs, odd x] == xs)
```

Las definiciones son equivalentes:

```
prop_equivalencia :: [Int] -> Bool
prop_equivalencia xs =
    todosImpares_2 xs == todosImpares_1 xs &&
    todosImpares_3 xs == todosImpares_1 xs
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

3.34. Triángulos numéricos

Ejercicio 3.34. *Definir la función* `triángulo` *tal que* `triángulo n` *es la lista de las lista de números consecutivos desde* `[1]` *hasta* `[1,2,...,n]`. *Por ejemplo,*

```
triángulo 4 ⇝ [[1],[1,2],[1,2,3],[1,2,3,4]]
```

Solución: Definición por comprensión:

```
triángulo :: Int -> [[Int]]
triángulo n = [[1..x] | x <- [1..n]]
```

3.35. Posición de un elemento en una lista

Ejercicio 3.35. *Definir la función* `posición` *tal que* `posición x ys` *es la primera posición del elemento* `x` *en la lista* `ys` *y* `0` *en el caso de que no pertenezca a la lista. Por ejemplo,*

```
posición 5 [1,5,3,5,6,5,3,4] ⇝ 2
```

Solución: Se presentan distintas definiciones:

1. Definición recursiva:

```
posición_1 :: Eq a => a -> [a] -> Int
posición_1 x ys =
    if elem x ys then aux x ys
    else 0
    where aux x [] = 0
          aux x (y:ys)
                | x== y     = 1
                | otherwise = 1 + aux x ys
```

2. Definición con listas de comprensión:

```
posición_2 :: Eq a => a -> [a] -> Int
posición_2 x xs = head ([pos |
                         (y,pos) <- zip xs [1..length xs],
                         y == x]
                        ++ [0])
```

Las definiciones son equivalentes:

```
prop_equivalencia :: Int -> [Int] -> Bool
prop_equivalencia x xs =
    posición_1 x xs == posición_2 x xs
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

Usaremos como `posición` la primera

```
posición:: Eq a => a -> [a] -> Int
posición = posición_1
```

Se verifica las siguiente propiedad: El elemento en la posición de `x` en `xs` es `x`:

```
prop_posición :: Int -> [Int] -> Bool
prop_posición x xs =
    let n=posición x xs in
    if n==0 then notElem x xs
    else xs!!(n-1)==x
```

3.36. Ordenación rápida

Ejercicio 3.36. *Definir la función* `ordenaR` *tal que* `ordenaR xs` *es la lista* `xs` *ordenada mediante el procedimiento de ordenación rápida. Por ejemplo,*

```
ordenaR [5,2,7,7,5,19,3,8,6] ⇝ [2,3,5,5,6,7,7,8,19]
```

Solución:

```
ordenaR :: Ord a => [a] -> [a]
ordenaR []     = []
ordenaR (x:xs) = ordenaR menores ++ [x] ++ ordenaR mayores
                 where menores = [e | e<-xs, e<x]
                       mayores = [e | e<-xs, e>=x]
```

El valor de `ordenaR` es una lista ordenada

```
prop_ordenaR_ordenada :: [Int] -> Bool
prop_ordenaR_ordenada xs =
    lista_ordenada (ordenaR xs)
```

```
Ordena_por_insercion> quickCheck prop_ordenaR_ordenada
OK, passed 100 tests.
```

3.37. Primera componente de un par

Ejercicio 3.37. *Redefinir la función* `fst` *tal que* `fst p` *es la primera componente del par* `p`*. Por ejemplo,*

```
fst (3,2) ⇝ 3
```

Solución:

```
n_fst :: (a,b) -> a
n_fst (x,_)  = x
```

Las definiciones son equivalentes:

```
prop_equivalencia :: (Int,Int) -> Bool
prop_equivalencia p =
    n_fst p == fst p
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

3.38. Segunda componente de un par

Ejercicio 3.38. *Redefinir la función* `snd` *tal que* `snd p` *es la segunda componente del par* `p`. *Por ejemplo,*

```
snd (3,2) ⇝ 2
```

Solución:

```
n_snd :: (a,b) -> b
n_snd (_,y)  = y
```

Las definiciones son equivalentes:

```
prop_equivalencia :: (Int,Int) -> Bool
prop_equivalencia p =
    n_snd p == snd p
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

3.39. Componentes de una terna

Ejercicio 3.39. *Redefinir las siguientes funciones*

- `fst3 t` *es la primera componente de la terna* `t`.
- `snd3 t` *es la segund componente de la terna* `t`.
- `thd3 t` *es la tercera componente de la terna* `t`.

Por ejemplo,

```
fst3 (3,2,5) ⇝ 3
snd3 (3,2,5) ⇝ 2
thd3 (3,2,5) ⇝ 5
```

Solución:

```
n_fst3         :: (a,b,c) -> a
n_fst3 (x,_,_)  = x

n_snd3         :: (a,b,c) -> b
n_snd3 (_,y,_)  = y

n_thd3         :: (a,b,c) -> c
n_thd3 (_,_,z)  = z
```

Se verifica la siguiente propiedad:

```
prop_ternas :: (Int,Int,Int) -> Bool
prop_ternas x =
    (n_fst3 x, n_snd3 x, n_thd3 x) == x
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

3.40. Creación de variables a partir de pares

Ejercicio 3.40. *Definir la función* `variable` *tal que* `variable p` *es la cadena correspondiente al par* `p` *formado por un carácter y un número. Por ejemplo,*

```
variable ('x',3) ⇝ "x3"
```

Solución:

```
variable :: (Char,Int) -> String
variable (c,n) = [c] ++ show n
```

3.41. División de una lista

Ejercicio 3.41. *Redefinir la función* `splitAt` *tal que* `splitAt n l` *es el par formado por la lista de los* `n` *primeros elementos de la lista* `l` *y la lista* `l` *sin los* `n` *primeros elementos. Por ejemplo,*

```
splitAt 3 [5,6,7,8,9,2,3] ⇝ ([5,6,7],[8,9,2,3])
splitAt 4 "sacacorcho"    ⇝ ("saca","corcho")
```

Solución:

```
n_splitAt :: Int -> [a] -> ([a], [a])
n_splitAt n xs | n <= 0 = ([],xs)
n_splitAt _ []          = ([],[])
n_splitAt n (x:xs)      = (x:xs',xs'')
    where (xs',xs'') = n_splitAt (n-1) xs
```

Se verifica la siguiente propiedad:

```
prop_splitAt :: Int -> [Int] -> Bool
prop_splitAt  n xs =
    n_splitAt n xs == (take n xs, drop n xs)
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

3.42. Sucesión de Fibonacci

Ejercicio 3.42. *Definir la función* `fib n` *tal que* `fib n` *es el* `n`*–ésimo término de la sucesión de Fibonacci 1,1,2,3,5,8,13,21,34,55,... Por ejemplo,*

```
fib 5  valor
```

Solución: Se presentan distintas definiciones:

1. Definición recursiva:

```
fib_1 :: Int -> Int
fib_1 0 = 1
fib_1 1 = 1
fib_1 (n+2) = fib_1 n + fib_1 (n+1)
```

2. Definición con acumuladores:

```
fib_2 :: Int -> Int
fib_2 n = fib_2_aux n 1 1
    where fib_2_aux 0 p q     = p
          fib_2_aux (n+1) p q = fib_2_aux n q (p+q)
```

3. Definición con mediante listas infinitas:

```
fib_3 :: Int -> Int
fib_3 n = fibs!!n
```

donde `fibs` es la sucesión de los números de Fibonacci.

```
fibs :: [Int]
fibs = 1 : 1 : [a+b | (a,b) <- zip fibs (tail fibs)]
```

Las definiciones son equivalentes:

```
prop_equivalencia :: Bool
prop_equivalencia =
    [fib_1 n | n <- [1..20]] == [fib_2 n | n <- [1..20]] &&
    [fib_3 n | n <- [1..20]] == [fib_2 n | n <- [1..20]]
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

La complejidad de `fib_1` es $O(fib(n))$ y la de `fib_2` y `fib_3` es $O(n)$, como se observa en la siguiente tabla donde se muestra el número de reducciones

n	fib_1	fib_2	fib_3
2	85	96	76
4	241	158	114
8	1.741	282	190
16	82.561	530	342
32	182.249.581	1.026	706

3.43. Incremento con el mínimo

Ejercicio 3.43. *Definir la función* `incmin` *tal que* `incmin l` *es la lista obtenida añadiendo a cada elemento de* `l` *el menor elemento de* `l`*. Por ejemplo,*

```
incmin [3,1,4,1,5,9,2,6] ⇝ [4,2,5,2,6,10,3,7]
```

Solución: Se presentan distintas definiciones:

1. Definición recursiva

```
incmin_1 :: [Int] -> [Int]
incmin_1 l = map (+e) l
             where e               = mínimo l
                   mínimo [x]      = x
                   mínimo (x:y:xs) = min x (mínimo (y:xs))
```

2. Con la definición anterir se recorre la lista dos veces: una para calcular el mínimo y otra para sumarlo. Con la siguiente definición la lista se recorre sólo una vez.

```
incmin_2 :: [Int] -> [Int]
incmin_2 [] = []
incmin_2 l = nuevalista
    where (minv, nuevalista) = un_paso l
          un_paso [x]        = (x, [x+minv])
          un_paso (x:xs)     = (min x y, (x+minv):ys)
                               where (y,ys) = un_paso xs
```

Las definiciones son equivalentes:

```
prop_equivalencia :: [Int] -> Bool
prop_equivalencia xs =
    incmin_1 xs == incmin_2 xs
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

3.44. Longitud de camino entre puntos bidimensionales

Ejercicio 3.44. *Definir el tipo* `Punto` *como un par de números reales Por ejemplo,*

```
(3.0,4.0) :: Punto
```

Solución:

```
type Punto = (Double, Double)
```

Ejercicio 3.45. *Definir la función* `distancia_al_origen` *tal que* `distancia_al_origen p` *es la distancia del punto* `p` *al origen. Por ejemplo,*

```
distancia_al_origen (3,4) ⇝ 5.0
```

Solución:

```
distancia_al_origen :: Punto -> Double
distancia_al_origen (x,y) = sqrt (x*x+y*y)
```

Ejercicio 3.46. *Definir la función* `distancia` *tal que* `distancia p1 p2` *es la distancia entre los puntos* `p1` *y* `p2`*. Por ejemplo,*

```
distancia (2,4) (5,8) ⇝ 5.0
```

Solución:

```
distancia :: Punto -> Punto -> Double
distancia (x,y) (x',y') = sqrt((x-x')^2+(y-y')^2)
```

Ejercicio 3.47. *Definir el tipo* `Camino` *como una lista de puntos Por ejemplo,*

```
[(1,2),(4,6),(7,10)] :: Camino
```

Solución:

```
type Camino = [Punto]
```

Ejercicio 3.48. *Definir la función* `longitud_camino` *tal que* `longitud_camino c` *es la longitud del camino* `c`*. Por ejemplo,*

```
longitud_camino [(1,2),(4,6),(7,10)] ⇝ 10.0
```

Solución: Se presentan distintas definiciones:

1. Definición recursiva:

```
longitud_camino_1 :: Camino -> Double
longitud_camino_1 (x:y:xs) = distancia x y + longitud_camino_1 (y:xs)
longitud_camino_1 _        = 0
```

2. Definición por comprensión:

```
longitud_camino_2 :: Camino -> Double
longitud_camino_2 xs =
    sum [distancia p q | (p,q) <- zip (init xs) (tail xs)]
```

Evaluación paso a paso:

```
longitud_camino_2 [(1,2),(4,6),(7,10)]
= sum [distancia p q | (p,q) <- zip (init [(1,2),(4,6),(7,10)])
                                    (tail [(1,2),(4,6),(7,10)])]
= sum [distancia p q | (p,q) <- zip [(1,2),(4,6)] [(4,6),(7,10)]]
= sum [distancia p q | (p,q) <- [((1,2),(4,6)),((4,6),(7,10))]]
= sum [5.0,5.0]
= 10
```

Las definiciones son equivalentes:

```
prop_equivalencia xs =
    not (null xs) ==>
    longitud_camino_1 xs ~= longitud_camino_2 xs

infix 4 ~=
(~=)   :: Double -> Double -> Bool
x ~= y  = abs(x-y) < 0.0001
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

3.45. Números racionales

Ejercicio 3.49. *Definir el tipo* `Racional` *de los números racionales como pares de enteros.*

Solución:

```
type Racional = (Int, Int)
```

Ejercicio 3.50. *Definir la función* `simplificar` *tal que* `simplificar x` *es el número racional* `x` *simplificado. Por ejemplo,*

```
simplificar (12,24)   ⇝ (1,2)
simplificar (12,-24)  ⇝ (-1,2)
simplificar (-12,-24) ⇝ (1,2)
simplificar (-12,24)  ⇝ (-1,2)
```

Solución:

```
simplificar (n,d) = (((signum d)*n) `div` m, (abs d) `div` m)
                    where m = gcd n d
```

Ejercicio 3.51. *Definir las operaciones entre números racionales* `qMul`, `qDiv`, `qSum` *y* `qRes`. *Por ejemplo,*

```
qMul (1,2) (2,3) ⇝ (1,3)
qDiv (1,2) (1,4) ⇝ (2,1)
qSum (1,2) (3,4) ⇝ (5,4)
qRes (1,2) (3,4) ⇝ (-1,4)
```

Solución:

```
qMul                 :: Racional -> Racional -> Racional
qMul (x1,y1) (x2,y2)  = simplificar (x1*x2, y1*y2)

qDiv                 :: Racional -> Racional -> Racional
qDiv (x1,y1) (x2,y2)  = simplificar (x1*y2, y1*x2)

qSum                 :: Racional -> Racional -> Racional
qSum (x1,y1) (x2,y2)  = simplificar (x1*y2+y1*x2, y1*y2)

qRes                 :: Racional -> Racional -> Racional
qRes (x1,y1) (x2,y2)  = simplificar (x1*y2-y1*x2, y1*y2)
```

Ejercicio 3.52. *Definir la función* `escribeRacional` *tal que* `escribeRacional x` *es la cadena correspodiente al número racional* `x`. *Por ejemplo,*

```
escribeRacional (10,12)            ⇝ "5/6"
escribeRacional (12,12)            ⇝ "1"
escribeRacional (qMul (1,2) (2,3)) ⇝ "1/3"
```

Solución:

```
escribeRacional :: Racional -> String
escribeRacional (x,y)
 | y' == 1 = show x'
 | otherwise = show x' ++ "/" ++ show y'
    where (x',y') = simplificar (x,y)
```

3.46. Máximo común divisor

Ejercicio 3.53. *Redefinir la función* `gcd` *tal que* `gcd x y` *es el máximo común divisor de* `x` *e* `y`. *Por ejemplo,*

```
gcd 6 15 ⇝ 3
```

Solución: Se presentan distintas definiciones:

1. Definición recursiva:

```
n_gcd_1 :: Int -> Int -> Int
n_gcd_1 0 0 = error "gcd 0 0 no está definido"
n_gcd_1 x y = n_gcd_1' (abs x) (abs y)
    where n_gcd_1' x 0 = x
          n_gcd_1' x y = n_gcd_1' y (x `rem` y)
```

2. Definición con `divisible` y `divisores`

```
n_gcd_2 :: Int -> Int -> Int
n_gcd_2 0 0 = error "gcd 0 0 no está definido"
n_gcd_2 0 y = abs y
n_gcd_2 x y = last (filter (divisible y') (divisores x'))
              where x'             = abs x
                    y'             = abs y
                    divisores x    = filter (divisible x) [1..x]
                    divisible x y  = x `rem` y == 0
```

Las definiciones son equivalentes:

```
prop_equivalencia :: Int -> Int -> Property
prop_equivalencia x y =
    (x,y) /= (0,0) ==>
    n_gcd_1 x y == gcd x y &&
    n_gcd_2 x y == gcd x y
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

3.47. Búsqueda en una lista de asociación

Ejercicio 3.54. *Redefinir la función* `lookup` *tal que* `lookup l z` *es el valor del primer elemento de la lista de búsqueda* `l` *cuya clave es* `z`*. Por ejemplo,*

```
lookup [('a',1),('b',2),('c',3),('b',4)] 'b' ⇝ 2
```

Solución:

```
n_lookup :: Eq a => a -> [(a,b)] -> Maybe b
n_lookup k []   = Nothing
n_lookup k ((x,y):xys)
    | k==x      = Just y
    | otherwise = n_lookup k xys
```

Las definiciones son equivalentes:

```
prop_equivalencia :: Int -> [(Int,Int)] -> Bool
prop_equivalencia z xys =
    n_lookup z xys == n_lookup z xys
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

Se verifica la siguiente propiedad

```
prop_lookup :: Int -> Int -> [(Int,Int)] -> Bool
prop_lookup x y xys =
    if n_lookup x xys == Just y then elem (x,y) xys
    else notElem (x,y) xys
```

Sin embargo, no es cierta la siguiente

```
prop_lookup_falsa :: Int -> Int -> [(Int,Int)] -> Bool
prop_lookup_falsa x y xys =
    if elem (x,y) xys then n_lookup x xys == Just y
    else n_lookup x xys == Nothing
```

En efecto,

```
Main> quickCheck prop_lookup_falsa
Falsifiable, after 0 tests:
-2
1
[(-2,-2)]
```

3.48. Emparejamiento de dos listas

Ejercicio 3.55. *Redefinir la función* `zip` *tal que* `zip x y` *es la lista obtenida emparejando los correspondientes elementos de* `x` *e* `y`*. Por ejemplo,*

```
zip [1,2,3] "abc" ⇝ [(1,'a'),(2,'b'),(3,'c')]
zip [1,2] "abc"   ⇝ [(1,'a'),(2,'b')]
```

Solución: Se presentan distintas definiciones:

1. Definición recursiva

```
n_zip_1 :: [a] -> [b] -> [(a,b)]
n_zip_1 (x:xs) (y:ys) = (x,y) : zip xs ys
n_zip_1 _      _      = []
```

2. Definición con `zipWith`

```
n_zip_2 :: [a] -> [b] -> [(a,b)]
n_zip_2 = zipWith  (\x y -> (x,y))
```

Las definiciones son equivalentes:

```
prop_equivalencia :: [Int] -> Bool
prop_equivalencia xs =
    n_zip_1 xs xs == zip xs xs &&
    n_zip_2 xs xs == zip xs xs
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

3.49. Emparejamiento funcional de dos listas

Ejercicio 3.56. *Redefinir la función* `zipWith` *tal que* `zipWith f x y` *es la lista obtenida aplicando la función* `f` *a los elementos correspondientes de las listas* `x` *e* `y`*. Por ejemplo,*

```
zipWith (+) [1,2,3] [4,5,6] ⇝ [5,7,9]
zipWith (*) [1,2,3] [4,5,6] ⇝ [4,10,18]
```

Solución:

```
n_zipWith :: (a->b->c) -> [a]->[b]->[c]
n_zipWith f (x:xs) (y:ys)  = f x y : zipWith f xs ys
n_zipWith _ _      _       = []
```

Las definiciones son equivalentes:

```
prop_equivalencia :: [Int] -> Bool
prop_equivalencia xs =
    n_zipWith (+) xs xs == zipWith (+) xs xs
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

3.50. Currificación

Ejercicio 3.57. *Una función está en* forma cartesiana *si su argumento es una tupla. Por ejemplo,*

```
suma_cartesiana       :: (Int,Int) -> Int
suma_cartesiana (x,y)  = x+y
```

En cambio, la función

```
suma_currificada     :: Int -> Int -> Int
suma_currificada x y  = x+y
```

está en forma currificada*.*

Redefinir la función `curry` *tal que* `curry f` *es la versión currificada de la función* `f`*. Por ejemplo,*

```
curry suma_cartesiana 2 3 ⇝ 5
```

y la función `uncurry` *tal que* `uncurry f` *es la versión cartesiana de la función* `f`*. Por ejemplo,*

```
uncurry suma_currificada (2,3) ⇝ 5
```

Solución:

```
n_curry :: ((a,b) -> c) -> (a -> b -> c)
n_curry f x y  = f (x,y)

n_uncurry :: (a -> b -> c) -> ((a,b) -> c)
n_uncurry f p  = f (fst p) (snd p)
```

3.51. Funciones sobre árboles

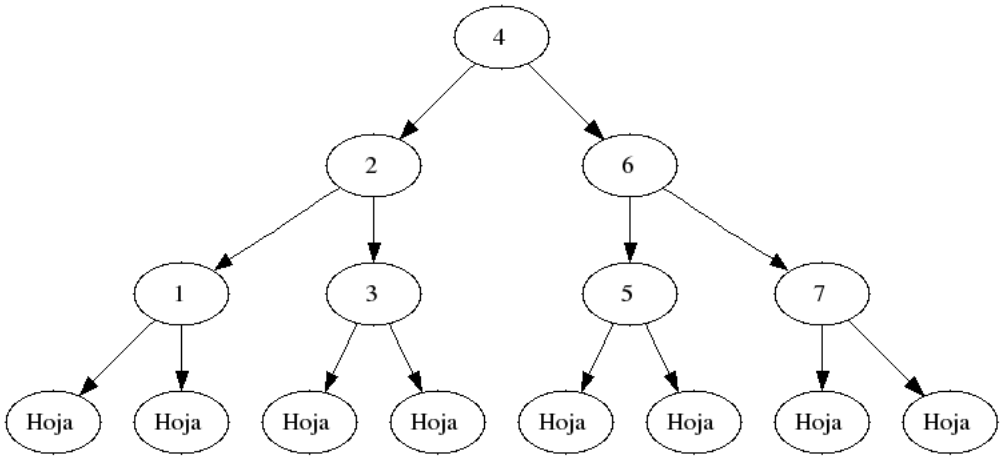
Ejercicio 3.58. *Un* árbol *de tipo* `a` *es una hoja de tipo* `a` *o es un nodo de tipo* `a` *con dos hijos que son árboles de tipo* `a`*.*

Definir el tipo `Árbol`*.*

Solución:

```
data Árbol a = Hoja
             | Nodo a (Árbol a) (Árbol a)
               deriving Show
```

Ejercicio 3.59. *Definir el árbol correspondiente a la siguiente figura*



Solución:

```
ejÁrbol_1 = Nodo 4 (Nodo 2 (Nodo 1 Hoja Hoja)
                           (Nodo 3 Hoja Hoja))
                   (Nodo 6 (Nodo 5 Hoja Hoja)
                           (Nodo 7 Hoja Hoja))
```

Ejercicio 3.60. *Definir la función* `tamaño` *tal que* `tamaño a` *es el tamaño del árbol* `a`*; es decir, el número de nodos internos. Por ejemplo,*

```
tamaño ejÁrbol_1 ⇝ 7
```

Solución:

```
tamaño :: Árbol a -> Int
tamaño Hoja            = 0
tamaño (Nodo x a1 a2)  = 1 + tamaño a1 + tamaño a2
```

Ejercicio 3.61. *Un* árbol de búsqueda *es un árbol binario en el que que todos los valores en el subárbol izquierdo son menores que el valor en el nodo mismo, y que todos los valores en el subárbol derecho son mayores. Por ejemplo, el* `ejÁrbol_1` *es un árbol de búsqueda.*

Definir la función `elemÁrbol` *tal que* `elemÁrbol e x` *se verifica si* `e` *es un elemento del árbol de búqueda* `x`*. Por ejemplo,*

```
elemÁrbol 5 ejÁrbol_1 ⇝ True
elemÁrbol 9 ejÁrbol_1 ⇝ False
```

Solución:

```
elemÁrbol :: Ord a => a -> Árbol a -> Bool
elemÁrbol e Hoja = False
elemÁrbol e (Nodo x izq der) | e==x = True
                             | e<x  = elemÁrbol e izq
                             | e>x  = elemÁrbol e der
```

Ejercicio 3.62. *Definir la función* `insertaÁrbol` *tal que* `insertaÁrbol e ab` *inserta el elemento* `e` *en el árbol de búsqueda* `ab`*. Por ejemplo,*

```
Main> insertaÁrbol 8 ejÁrbol_1
Nodo 4 (Nodo 2
             (Nodo 1 Hoja Hoja)
             (Nodo 3 Hoja Hoja))
       (Nodo 6
             (Nodo 5 Hoja Hoja)
```

```
             (Nodo 7
                   Hoja
                   (Nodo 8 Hoja Hoja)))
Main> insertaÁrbol 3 ejÁrbol_1
Nodo 4 (Nodo 2
             (Nodo 1 Hoja Hoja)
             (Nodo 3
                   (Nodo 3 Hoja Hoja)
                   Hoja))
       (Nodo 6
             (Nodo 5 Hoja Hoja)
             (Nodo 7 Hoja Hoja))
```

Solución:

```
insertaÁrbol :: Ord a => a -> Árbol a -> Árbol a
insertaÁrbol e Hoja = Nodo e Hoja Hoja
insertaÁrbol e (Nodo x izq der)
    | e <= x = Nodo x (insertaÁrbol e izq) der
    | e > x  = Nodo x izq (insertaÁrbol e der)
```

Ejercicio 3.63. *Definir la función* `listaÁrbol` *tal que* `listaÁrbol l` *es el árbol de búsqueda obtenido a partir de la lista* `l`*. Por ejemplo,*

```
Main> listaÁrbol [3,2,4,1]
Nodo 1
     Hoja
     (Nodo 4
           (Nodo 2
                 Hoja
                 (Nodo 3 Hoja Hoja))
           Hoja)
```

Solución:

```
listaÁrbol :: Ord a => [a] -> Árbol a
listaÁrbol = foldr insertaÁrbol Hoja
```

Ejercicio 3.64. *Definir la función* `aplana` *tal que* `aplana ab` *es la lista obtenida aplanando el árbol* `ab`*. Por ejemplo,*

```
aplana ejÁrbol_1              ⇝ [1,2,3,4,5,6,7]
aplana (listaÁrbol [3,2,4,1]) ⇝ [1,2,3,4]
```

Solución:

```
aplana :: Árbol a -> [a]
aplana Hoja = []
aplana (Nodo x izq der) = aplana izq ++ [x] ++ aplana der
```

Ejercicio 3.65. *Definir la función* `ordenada_por_árbol` *tal que* `ordenada_por_árbol l` *es la lista* `l` *ordenada mediante árbol de búsqueda. Por ejemplo,*

```
ordenada_por_árbol [1,4,3,7,2] ⇝ [1,2,3,4,7]
```

Solución:

```
ordenada_por_árbol :: Ord a => [a] -> [a]
ordenada_por_árbol = aplana . listaÁrbol
```

Se verifica la siguiente propiedad

```
prop_ordenada_por_árbol :: [Int] -> Bool
prop_ordenada_por_árbol xs =
    lista_ordenada (ordenada_por_árbol xs)
```

En efecto,

```
Main> quickCheck prop_ordenada_por_arbol
OK, passed 100 tests.
```

3.52. Búsqueda en lista ordenada

Ejercicio 3.66. *Definir la función* `elem_ord` *tal que* `elem_ord e l` *se verifica si* `e` *es un elemento de la lista ordenada* `l`*. Por ejemplo,*

```
elem_ord 3 [1,3,5] ⇝ True
elem_ord 2 [1,3,5] ⇝ False
```

Solución:

```
elem_ord :: Ord a => a -> [a] -> Bool
elem_ord _ []                  = False
elem_ord e (x:xs) | x < e      = elem_ord e xs
                  | x == e     = True
                  | otherwise  = False
```

3.53. Movimiento según las direcciones

Ejercicio 3.67. *Definir el tipo finito* `Direcciόm` *tal que sus constructores son Norte, Sur, Este y Oeste.*

Solución:

```
data Dirección = Norte | Sur | Este | Oeste
```

Ejercicio 3.68. *Definir la función* `mueve` *tal que* `mueve d p` *es el punto obtenido moviendo el punto* `p` *una unidad en la dirección* `d`*. Por ejemplo,*

```
mueve Sur (mueve Este (4,7)) ⇝ (5,6)
```

Solución:

```
mueve :: Dirección -> (Int,Int) -> (Int,Int)
mueve Norte (x,y) = (x,y+1)
mueve Sur   (x,y) = (x,y-1)
mueve Este  (x,y) = (x+1,y)
mueve Oeste (x,y) = (x-1,y)
```

3.54. Los racionales como tipo abstracto de datos

Ejercicio 3.69. *Definir el tipo de datos* `Ratio` *para representar los racionales como un par de enteros (su numerador y denominador).*

Solución:

```
data Ratio =  Rac Int Int
```

Ejercicio 3.70. *Definir* `Ratio` *como una instancia de* `Show` *de manera que la función* `show` *muestra la forma simplificada obtenida mediante la función* `simplificarRatio` *tal que* `simplificarRatio x` *es el número racional* `x` *simplificado. Por ejemplo,*

```
simplificarRatio (Rac  12  24) ⇝  1/2
simplificarRatio (Rac  12 -24) ⇝ -1/2
simplificarRatio (Rac -12 -24) ⇝  1/2
simplificarRatio (Rac -12  24) ⇝ -1/2
```

Solución:

```
instance Show Ratio where
    show (Rac x 1) = show x
    show (Rac x y) = show x' ++ "/" ++ show y'
                     where (Rac x' y') = simplificarRatio (Rac x y)

simplificarRatio :: Ratio -> Ratio
simplificarRatio (Rac n d) = Rac (((signum d)*n) `div` m) ((abs d) `div` m)
                             where m = gcd n d
```

Ejercicio 3.71. *Definir los números racionales 0, 1, 2, 3, 1/2, 1/3 y 1/4. Por ejemplo,*

```
Main> :set +t
Main> rDos
2 :: Ratio
Main> rTercio
1/3 :: Ratio
```

Solución:

```
rCero   = Rac 0 1
rUno    = Rac 1 1
rDos    = Rac 2 1
rTres   = Rac 3 1
rMedio  = Rac 1 2
rTercio = Rac 1 3
rCuarto = Rac 1 4
```

Ejercicio 3.72. *Definir las operaciones entre números racionales* `rMul`, `rDiv`, `rSum` *y* `rRes`. *Por ejemplo,*

```
rMul (Rac 1 2) (Rac 2 3) ⇝  1/3
rDiv (Rac 1 2) (Rac 1 4) ⇝  2
rSum (Rac 1 2) (Rac 3 4) ⇝  5/4
rRes (Rac 1 2) (Rac 3 4) ⇝ -1/4
```

Solución:

```
rMul                    :: Ratio -> Ratio -> Ratio
rMul (Rac a b) (Rac c d)  = simplificarRatio (Rac (a*c) (b*d))

rDiv                    :: Ratio -> Ratio -> Ratio
rDiv (Rac a b) (Rac c d)  = simplificarRatio (Rac (a*d) (b*c))
```

```
rSum                        :: Ratio -> Ratio -> Ratio
rSum (Rac a b) (Rac c d)  = simplificarRatio (Rac (a*d+b*c) (b*d))

rRes                        :: Ratio -> Ratio -> Ratio
rRes (Rac a b) (Rac c d)  = simplificarRatio (Rac (a*d-b*c) (b*d))
```

Capítulo 4

Aplicaciones de programación funcional

Contenido

4.1. Segmentos iniciales

Ejercicio 4.1. *Definir la función* `iniciales` *tal que* `iniciales l` *es la lista de los segmentos iniciales de la lista* `l`*. Por ejemplo,*

```
iniciales [2,3,4]   ⇝ [[],[2],[2,3],[2,3,4]]
iniciales [1,2,3,4] ⇝ [[],[1],[1,2],[1,2,3],[1,2,3,4]]
```

Solución:

```
iniciales :: [a] -> [[a]]
iniciales []     = [[]]
iniciales (x:xs) = [] : [x:ys | ys <- iniciales xs]
```

El número de los segmentos iniciales es el número de los elementos de la lista más uno.

```
prop_iniciales :: [Int] -> Bool
prop_iniciales xs =
    length(iniciales xs) == 1 + length xs
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

4.2. Segmentos finales

Ejercicio 4.2. *Definir la función* `finales` *tal que* `finales l` *es la lista de los segmentos finales de la lista* `l`*. Por ejemplo,*

```
finales [2,3,4]   ⇝ [[2,3,4],[3,4],[4],[]]
finales [1,2,3,4] ⇝ [[1,2,3,4],[2,3,4],[3,4],[4],[]]
```

Solución:

```
finales :: [a] -> [[a]]
finales []     = [[]]
finales (x:xs) = (x:xs) : finales xs
```

El número de los segmentos finales es el número de los elementos de la lista más uno.

```
prop_finales :: [Int] -> Bool
prop_finales xs =
    length(finales xs) == 1 + length xs
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

4.3. Segmentos

Ejercicio 4.3. *Definir la función* `segmentos` *tal que* `segmentos l` *es la lista de los segmentos de la lista* `l`*. Por ejemplo,*

```
Main> segmentos [2,3,4]
[[],[4],[3],[3,4],[2],[2,3],[2,3,4]]
Main> segmentos [1,2,3,4]
[[],[4],[3],[3,4],[2],[2,3],[2,3,4],[1],[1,2],[1,2,3],[1,2,3,4]]
```

Solución:

```
segmentos :: [a] -> [[a]]
segmentos []     = [[]]
segmentos (x:xs) = segmentos xs ++ [x:ys | ys <- iniciales xs]
```

4.4. Sublistas

Ejercicio 4.4. *Definir la función* `sublistas` *tal que* `sublistas l` *es la lista de las sublistas de la lista* `l`*. Por ejemplo,*

```
Main> sublistas [2,3,4]
[[2,3,4],[2,3],[2,4],[2],[3,4],[3],[4],[]]
Main> sublistas [1,2,3,4]
[[1,2,3,4],[1,2,3],[1,2,4],[1,2],[1,3,4],[1,3],[1,4],[1],
 [2,3,4],    [2,3],  [2,4],  [2],  [3,4],  [3],  [4], []]
```

Solución:

```
sublistas :: [a] -> [[a]]
sublistas []     = [[]]
sublistas (x:xs) = [x:ys | ys <- sub] ++ sub
                   where sub = sublistas xs
```

4.5. Comprobación de subconjunto

Ejercicio 4.5. *Definir la función* `subconjunto` *tal que* `subconjunto xs ys` *se verifica si* `xs` *es un subconjunto de* `ys`*. Por ejemplo,*

```
subconjunto [1,3,2,3] [1,2,3] ⇝ True
subconjunto [1,3,4,3] [1,2,3] ⇝ False
```

Solución: Se presentan distintas definiciones:

1. Definición recursiva:

```
subconjunto_1 :: Eq a => [a] -> [a] -> Bool
subconjunto_1 []     _ = True
subconjunto_1 (x:xs) ys = elem x ys && subconjunto_1 xs ys
```

2. Definición mediante `all`:

```
subconjunto_2 :: Eq a => [a] -> [a] -> Bool
subconjunto_2 xs ys = all (`elem` ys) xs
```

3. Usaremos como `subconjunto` la primera

```
subconjunto :: Eq a => [a] -> [a] -> Bool
subconjunto = subconjunto_1
```

Las definiciones son equivalentes:

```
prop_equivalencia :: [Int] -> [Int] -> Bool
prop_equivalencia xs ys =
    subconjunto_1 xs ys == subconjunto_2 xs ys
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

4.6. Comprobación de la igualdad de conjuntos

Ejercicio 4.6. *Definir la función* `igual_conjunto` *tal que* `igual_conjunto l1 l2` *se verifica si las listas* `l1` *y* `l2` *vistas como conjuntos son iguales Por ejemplo,*

```
igual_conjunto [1..10] [10,9..1]  ⇝ True
igual_conjunto [1..10] [11,10..1] ⇝ False
```

Solución: Se presentan distintas definiciones:

1. Usando `subconjunto`

```
igual_conjunto_1 :: Eq a => [a] -> [a] -> Bool
igual_conjunto_1 xs ys = subconjunto xs ys && subconjunto ys xs
```

2. Por recursión.

```
igual_conjunto_2 :: Eq a => [a] -> [a] -> Bool
igual_conjunto_2 xs ys = aux (nub xs) (nub ys)
    where aux []    []    = True
          aux (x:_) []    = False
          aux []    (y:_) = False
          aux (x:xs) ys   = x `elem` ys && aux xs (delete x ys)
```

3. Usando `sort`

```
igual_conjunto_3 :: (Eq a, Ord a) => [a] -> [a] -> Bool
igual_conjunto_3 xs ys = sort (nub xs) == sort (nub ys)
```

4. Usaremos como `igual_conjunto` la primera

```
igual_conjunto :: Eq a => [a] -> [a] -> Bool
igual_conjunto = igual_conjunto_1
```

Las definiciones son equivalentes:

```
prop_equivalencia :: [Int] -> [Int] -> Bool
prop_equivalencia xs ys =
    igual_conjunto_2 xs ys == igual_conjunto_1 xs ys &&
    igual_conjunto_3 xs ys == igual_conjunto_1 xs ys
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

4.7. Permutaciones

Ejercicio 4.7. *Definir la función* `permutaciones` *tal que* `permutaciones l` *es la lista de las permutaciones de la lista* `l`*. Por ejemplo,*

```
Main> permutaciones [2,3]
[[2,3],[3,2]]
Main> permutaciones [1,2,3]
[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
```

Solución: Se presentan distintas definiciones:

1. Por elección y recursión:

```
import Data.List ((\\))

permutaciones_1 :: Eq a => [a] -> [[a]]
permutaciones_1 [] = [[]]
permutaciones_1 xs = [a:p | a <- xs, p <- permutaciones_1(xs \\ [a])]
```

2. Por recursión e intercalación:

```
permutaciones_2 :: [a] -> [[a]]
permutaciones_2 []     = [[]]
permutaciones_2 (x:xs) = [zs | ys <- permutaciones_2 xs,
                               zs <- intercala x ys]
```

donde `intercala x ys` es la lista de las listas obtenidas intercalando `x` entre los elementos de la lista `l`. Por ejemplo,

```
intercala 1 [2,3] ⇝ [[1,2,3],[2,1,3],[2,3,1]]
```

```
intercala :: a -> [a] -> [[a]]
intercala e []     = [[e]]
intercala e (x:xs) = (e:x:xs) : [(x:ys) | ys <- (intercala e xs)]
```

Las definiciones son equivalentes:

```
prop_equivalencia :: [Int] -> Property
prop_equivalencia xs =
    length xs <= 6 ==>
    igual_conjunto (permutaciones_1 xs) (permutaciones_2 xs)
```

Comprobación

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

El número de permutaciones de un conjunto de `n` elementos es el factorial de `n`.

```
prop_número_permutaciones :: [Int] -> Property
prop_número_permutaciones xs =
    length xs <= 6 ==>
    length (permutaciones_2 xs) == factorial (length xs)
    where factorial n = product [1..n]
```

En la propiedades hemos acotado la longitude máxima de las listas generadas para facilitar los cálculos.

La segunda definición es más eficiente:

n	permutaciones_1	permutaciones_2
2	140	102
3	334	172
4	1.170	428
5	5.656	1.740
6	34.192	10.036
7	243.744	71.252

donde las columnas segunda y tercera contiene el número de reducciones.

4.8. Combinaciones

Ejercicio 4.8. *Definir la función* `combinaciones` *tal que* `combinaciones n l` *es la lista de las combinaciones* `n`*–arias de la lista* `l`*. Por ejemplo,*

```
combinaciones 2 [1,2,3,4] ⇝ [[1,2],[1,3],[1,4],[2,3],[2,4],[3,4]]
```

Solución: Se presentan distintas definiciones:

1. Definición mediante `sublistas`:

```
combinaciones_1 :: Int -> [a] -> [[a]]
combinaciones_1 n xs = [ys | ys <- sublistas xs, length ys == n]
```

2. Definición directa:

```
combinaciones_2 :: Int -> [a] -> [[a]]
combinaciones_2 0 _          = [[]]
combinaciones_2 _ []         = []
combinaciones_2 (n+1) (x:xs) = [x:ys | ys <- combinaciones_2 n xs] ++
                               combinaciones_2 (n+1) xs
```

La segunda definición es más eficiente como se comprueba en la siguiente sesión

```
Main> :set +s
Main> length (combinaciones_1 2 [1..15])
105
(1917964 reductions, 2327983 cells, 3 garbage collections)
Main> length (combinaciones_2 2 [1..15])
105
(6217 reductions, 9132 cells)
```

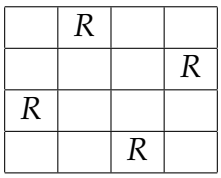
4.9. El problema de las reinas

Ejercicio 4.9. *El problema de las N reinas consiste en colocar N reinas en un tablero rectangular de dimensiones N por N de forma que no se encuentren más de una en la misma línea: horizontal, vertical o diagonal.*

Definir la función `reinas` *tal que* `reinas n` *es la lista de las soluciones del problema de las N reinas. Por ejemplo,*

```
reinas 4 ⇝ [[3,1,4,2],[2,4,1,3]]
```

La primera solución `[3,1,4,2]` *se interpreta como*



Solución: Se importa la diferencia de conjuntos (`\\`) del módulo `List`:

```
import Data.List ((\\))
```

El tablero se representa por una lista de números que indican las filas donde se han colocado las reinas. Por ejemplo, `[3,5]` indica que se han colocado las reinas `(1,3)` y `(2,5)`.

```
type Tablero = [Int]
```

La definición de `reinas` es

```
reinas :: Int -> [Tablero]
reinas n = reinasAux n
           where reinasAux 0     = [[]]
                 reinasAux (m+1) = [r:rs | rs <- reinasAux m,
                                           r <- ([1..n] \\ rs),
                                           noAtaca r rs 1]
```

donde `noAtaca r rs d` se verifica si la reina `r` no ataca a niguna de las de la lista `rs` donde la primera de la lista está a una distancia horizontal `d`.

```
noAtaca :: Int -> Tablero -> Int -> Bool
noAtaca _ [] _ = True
noAtaca r (a:rs) distH = abs(r-a) /= distH &&
                         noAtaca r rs (distH+1)
```

4.10. Números de Hamming

Ejercicio 4.10. *Los números de Hamming forman una sucesión estrictamente creciente de números que cumplen las siguientes condiciones:*

1. *El número 1 está en la sucesión.*
2. *Si x está en la sucesión, entonces $2 \times x$, $3 \times x$ y $5 \times x$ también están.*
3. *Ningún otro número está en la sucesión.*

Definir la función `hamming` *tal que* `hamming` *es la sucesión de Hamming. Por ejemplo,*

```
take 15 hamming ⇝ [1,2,3,4,5,6,8,9,10,12,15,16,18,20,24]
```

Solución:

```
hamming :: [Int]
hamming = 1 : mezcla3 [2*i | i <- hamming]
                      [3*i | i <- hamming]
                      [5*i | i <- hamming]
```

donde `mezcla3 xs ys zs` es la lista obtenida mezclando las listas ordenadas `xs`, `ys` y `zs` y eliminando los elementos duplicados. Por ejemplo,

```
mezcla3 [2,4,6,8,10] [3,6,9,12] [5,10] ⇝ [2,3,4,5,6,8,9,10,12]
```

```
mezcla3 :: [Int] -> [Int] -> [Int] -> [Int]
mezcla3 xs ys zs = mezcla2 xs (mezcla2 ys zs)
```

y `mezcla2 xs ys zs` es la lista obtenida mezclando las listas ordenadas `xs` e `ys` y eliminando los elementos duplicados. Por ejemplo,

```
mezcla2 [2,4,6,8,10,12] [3,6,9,12] ⇝ [2,3,4,6,8,9,10,12]
```

```
mezcla2 :: [Int] -> [Int] -> [Int]
mezcla2 p@(x:xs) q@(y:ys) | x < y     = x:mezcla2 xs q
                          | x > y     = y:mezcla2 p  ys
                          | otherwise = x:mezcla2 xs ys
mezcla2 []       ys                   = ys
mezcla2 xs       []                   = xs
```

Bibliografía

[1] H. C. Cunningham. Notes on functional programming with haskell. Technical report, University of Mississippi, 2007.

[2] J. Fokker. Programación funcional. Technical report, Universidad de Utrech, 1996.

[3] B. C. Ruiz, F. Gutiérrez, P. Guerrero, and J. Gallardo. *Razonando con Haskell (Un curso sobre programación funcional)*. Thompson, 2004.

[4] S. Thompson. *Haskell: The Craft of Functional Programming*. Addison–Wesley, second edition, 1999. En Bib.

[5] E. P. Wentworth. Introduction to Funcional Programming. Technical report, Parallel Processing Group. Rhodes University, 1994.

Índice alfabético