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 1 de agosto de 2008)

Índice general

Parte I

Programación básica

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
```

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' ⇝ "aaaaaaaaaaaaaaaaaaaaaaaaaa{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)
```

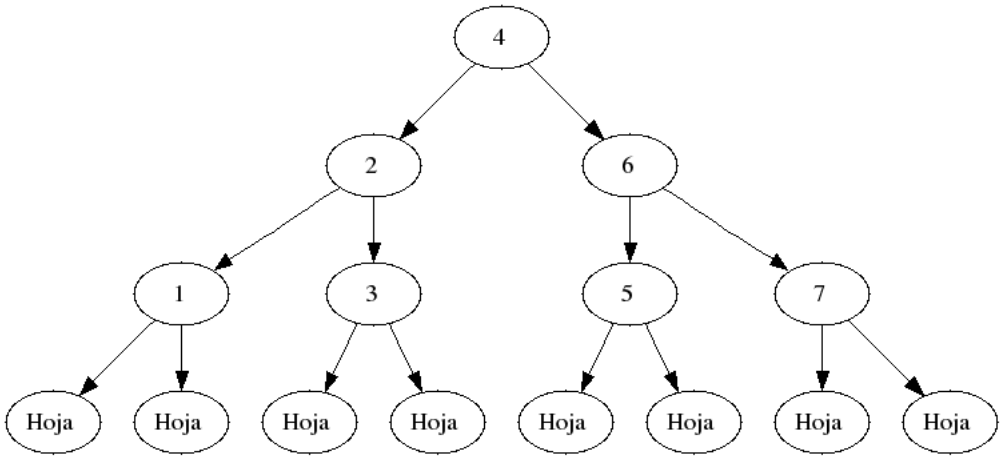
3.51. Funciones sobre árboles

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*

	R		
			R
R			
		R	

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
```

Parte II

Ejercicios del curso de K.L. Claessen
Introduction to Functional Programming

Capítulo 5

Introducción a la programación funcional

Contenido

Estos ejercicios corresponden a la primera clase.

5.1. Transformación entre euros y pesetas

Ejercicio 5.1. *El objetivo del ejercicio es presentar la manera de definir funciones aritméricas y comprobar propiedades usando QuickCheck.*

Ejercicio 5.1.1. *Calcular cuántas pesetas son 49 euros (1 euro son 166.386 pesetas).*

Solución: El cálculo es

```
Hugs> 49*cambioEuro
8152.914
```

Ejercicio 5.1.2. *Definir la constante* `cambioEuro` *cuyo valor es 166.386 y repetir el cálculo anterior usando la constante definida.*

Solución: La definición es

```
cambioEuro = 166.386
```

y el cálculo es

```
Main> 49
8152.914
```

Ejercicio 5.1.3. *Definir la función* `pesetas` *tal que* `pesetas x` *es la cantidad de pesetas correspondientes a* `x` *euros y repetir el cálculo anterior usando la función definida.*

Solución: La definición es

```
pesetas x = x*cambioEuro
```

y el cálculo es

```
Main> pesetas 49
8152.914
```

Ejercicio 5.1.4. *Definir la función* `euros` *tal que* `euros x` *es la cantidad de euros correspondientes a* `x` *pesetas y calcular los euros correspondientes a 8152.914 pesetas.*

Solución: La definición es

```
euros x = x/cambioEuro
```

y el cálculo es

```
Main> euros 8152.914
49.0
```

Ejercicio 5.1.5. *Definir la propiedad* `prop_EurosPesetas` *tal que* `prop_EurosPesetas x` *se verifique si al transformar* `x` *euros en pesetas y las pesetas obtenidas en euros se obtienen* `x` *euros. Comprobar la* `prop_EurosPesetas` *con 49 euros.*

Solución: La definición es

```
prop_EurosPesetas x =
    euros(pesetas x) == x
```

y la comprobación es

```
Main> prop_EurosPesetas 49
True
```

Ejercicio 5.1.6. *Comprobar la* `prop_EurosPesetas` *con QuickCheck.*

Solución: Para usar QuickCheck hay que importarlo escribiendo, al comienzo del fichero,

```
import Test.QuickCheck
```

La comprobación es

```
Main> quickCheck prop_EurosPesetas
Falsifiable, after 42 tests:
3.625
```

lo que indica que no se cumple para 3.625.

Ejercicio 5.1.7. *Calcular la diferencia entre euros(pesetas 3.625) y 3.625.*

Solución: El cálculo es

```
Main> euros(pesetas 3.625)-3.625
-4.44089209850063e-16
```

Ejercicio 5.1.8. *Se dice que* `x` *e* `y` *son casi iguales si el valor absoluto de la diferencia entre* `x` *e* `y` *es menor que una milésima. Definir el operador* `~=` *tal que* `x ~= y` *se verifique si* `x` *e* `y` *son casi iguales.*

Solución: La definición es

```
x ~= y = abs(x-y)<0.001
```

Ejercicio 5.1.9. *Definir la propiedad* `prop_EurosPesetas'` *tal que* `prop_EurosPesetas' x` *se verifique si al transformar* `x` *euros en pesetas y las pesetas obtenidas en euros se obtiene una cantidad casi igual a* `x` *de euros. Comprobar la* `prop_EurosPesetas'` *con 49 euros.*

Solución: La definición es

```
prop_EurosPesetas' x =
    euros(pesetas x) ~= x
```

y la comprobación es

```
Main> prop_EurosPesetas' 49
True
```

Ejercicio 5.1.10. *Comprobar la* `prop_EurosPesetas'` *con QuickCheck.*

Solución: La comprobación es

```
Main> quickCheck prop_EurosPesetas'
OK, passed 100 tests.
```

lo que indica que se cumple para los 100 casos de pruebas considerados.

5.2. Cuadrado

Ejercicio 5.2. *Definir la función*

```
cuadrado :: Integer -> Integer
```

tal que `(cuadrado x)` *es el cuadrado del número* `x`*. Por ejemplo,*

```
cuadrado 3  ⇝  9
```

Solución: La definición es

```
cuadrado :: Integer -> Integer
cuadrado x = x*x
```

5.3. Valor absoluto

Ejercicio 5.3. *Redefinir la función*

```
abs :: Integer -> Integer
```

tal que `(abs x)` *es el valor absoluto de* `x`*. Por ejemplo,*

```
abs (-3) ⇝ 3
abs 3    ⇝ 3
```

Solución: La definición, usando condicionales, es

```
n_abs_1 :: Integer -> Integer
n_abs_1 x = if x>0 then x else (-x)
```

La definición, usando guardas, es

```
n_abs_2 :: Integer -> Integer
n_abs_2 x | x>0       = x
          | otherwise = -x
```

5.4. Potencia

Ejercicio 5.4. *Definir la función*

```
potencia :: Integer -> Integer -> Integer
```

tal que `(potencia x y)` *es* x^y. *Por ejemplo,*

```
potencia 2 4   ⇝ 16
```

Solución: La definición es

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

5.5. Regiones en el plano

Ejercicio 5.5. *Definir la función*

```
regiones :: Integer -> Integer
```

tal que `(regiones n)` *es el número máximo de regiones en el plano generadas con* `n` *líneas. Por ejemplo,*

```
regiones 3  ⇝  7
```

Solución: La definición es

```
regiones :: Integer -> Integer
regiones 0       = 1
regiones n | n>0 = regiones (n-1) + n
```

Capítulo 6

Modelización y tipos de datos

Contenido

Estos ejercicios corresponden a la segunda clase.

6.1. Modelización de un juego de cartas

Ejercicio 6.1.1. *Definir el tipo de datos* `Palo` *para representar los cuatro palos de la baraja: picas, corazones, diamantes y tréboles. Hacer que* `Palo` *sea instancia de* `Eq` *y* `Show`.

Solución: La definición es

```
data Palo = Picas | Corazones | Diamantes | Tréboles
            deriving (Eq, Show)
```

Ejercicio 6.1.2. *Consultar la información sobre el tipo de datos* `Palo`.

Solución: La consulta es

```
Main> :i Palo
-- type constructor
data Palo

-- constructors:
```

```
Picas :: Palo
Corazones :: Palo
Diamantes :: Palo
Tréboles :: Palo
```

Ejercicio 6.1.3. *Consultar la información sobre el constructor* `Picas`.

Solución: La consulta es

```
Main> :i Picas
Picas :: Palo  -- data constructor
```

Ejercicio 6.1.4 (Avanzado)**.** *Definir un generador de* `Palos` *para QuickCheck.*

Solución:

```
instance Arbitrary Palo where
    arbitrary = elements [Picas, Corazones, Diamantes, Tréboles]
```

Ejercicio 6.1.5. *Definir el tipo de dato* `Color` *para representar los colores de las cartas: rojo y negro.*

Solución:

```
data Color = Rojo | Negro
             deriving Show
```

Ejercicio 6.1.6. *Definir la función*

```
color :: Palo -> Color
```

tal que `(color p)` *es el color del palo* `p`. *Por ejemplo,*

```
color Corazones  ⇝  Rojo
```

Solución:

```
color :: Palo -> Color
color Picas     = Negro
color Corazones = Rojo
color Diamantes = Rojo
color Tréboles  = Negro
```

Ejercicio 6.1.7. *Los valores de las cartas se dividen en los numéricos (del 2 al 10) y las figuras (sota, reina, rey y as). Definir el tipo de datos* `Valor` *para representar los valores de las cartas. Hacer que* `Valor` *sea instancia de* `Eq` *y* `Show`.

Solución:

```
data Valor = Numérico Int | Sota | Reina | Rey | As
             deriving (Eq, Show)
```

Ejercicio 6.1.8. *Consultar la información sobre el constructor Numérico.*

Solución: La consulta es

```
Main> :i Numérico
Numérico :: Int -> Valor  -- data constructor
```

Ejercicio 6.1.9. *Calcular el valor de* `(Numérico 3)`.

Solución: El cálculo es

```
Main> Numérico 3
Numérico 3
```

Ejercicio 6.1.10 (Avanzado)**.** *Definir un generador de valores para QuickCheck.*

Solución:

```
instance Arbitrary Valor where
  arbitrary =
    oneof $
      [ do return c
      | c <- [Sota,Reina,Rey,As]
      ] ++
      [ do n <- choose (2,10)
           return (Numérico n)
      ]
```

Ejercicio 6.1.11. *El orden de valor de las cartas (de mayor a menor) es as, rey, reina, sota y las numéricas según su valor. Definir la función*

```
mayor :: Valor -> Valor -> Bool
```

tal que `(mayor x y)` *se verifica si la carta* `x` *es de mayor valor que la carta* `y`*. Por ejemplo,*

```
mayor Sota (Numérico 7)    ⇝  True
mayor (Numérico 10) Reina  ⇝  False
```

Solución:

```
mayor :: Valor -> Valor -> Bool
mayor _            As           = False
mayor As           _            = True
mayor _            Rey          = False
mayor Rey          _            = True
mayor _            Reina        = False
mayor Reina        _            = True
mayor _            Sota         = False
mayor Sota         _            = True
mayor (Numérico m) (Numérico n) = m > n
```

Ejercicio 6.1.12. *Comprobar con QuickCheck si dadas dos cartas, una siempre tiene mayor valor que la otra.*

Solución: La propiedad es

```
prop_MayorValor1 a b =
    mayor a b || mayor b a
```

La comprobación es

```
Main> quickCheck prop_MayorValor1
Falsifiable, after 2 tests:
Sota
Sota
```

que indica que la propiedad es falsa porque la sota no tiene mayor valor que la sota.

Ejercicio 6.1.13. *Comprobar con QuickCheck si dadas dos cartas distintas, una siempre tiene mayor valor que la otra.*

Solución: La propiedad es

```
prop_MayorValor a b =
    a /= b ==> mayor a b || mayor b a
```

La comprobación es

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

Ejercicio 6.1.14. *Definir el tipo de datos* `Carta` *para representar las cartas mediante un valor y un palo.*

Solución:

```
data Carta = Carta Valor Palo
             deriving (Eq, Show)
```

Ejercicio 6.1.15. *Definir la función*

```
valor :: Carta -> Valor
```

tal que `(valor c)` *es el valor de la carta* `c`*. Por ejemplo,*

```
valor (Carta Rey Corazones)  ⇝  Rey
```

Solución:

```
valor :: Carta -> Valor
valor (Carta v p) = v
```

Ejercicio 6.1.16. *Definir la función*

```
palo :: Carta -> Valor
```

tal que `(palo c)` *es el palo de la carta* `c`*. Por ejemplo,*

```
palo (Carta Rey Corazones)  ⇝  Corazones
```

Solución:

```
palo :: Carta -> Palo
palo (Carta v p) = p
```

Ejercicio 6.1.17. *Una forma alternativa consiste en definir junto al tipo las funciones de acceso. Redefinir el tipo Carta1 de esta forma.*

Solución:

```
data Carta1 = Carta1 {valor1 :: Valor, palo1 :: Palo}
              deriving Show
```

Ejercicio 6.1.18. *Calcular*

- `(valor1 (Carta1 Rey Corazones))`
- `(palo1 (Carta1 Rey Corazones))`

Solución: El cálculo es

```
Main> valor1 (Carta1 Rey Corazones)
Rey
Main> palo1 (Carta1 Rey Corazones)
Corazones
```

Ejercicio 6.1.19 (Avanzado). *Definir un generador de cartas para QuickCheck.*

Solución:

```
instance Arbitrary Carta where
    arbitrary =
        do v <- arbitrary
           p <- arbitrary
           return (Carta v p)
```

Ejercicio 6.1.20. *Definir la función*

```
ganaCarta :: Palo -> Carta -> Carta -> Bool
```

tal que `(ganaCarta p c1 c2)` *se verifica si la carta* `c1` *le gana a la carta* `c2` *cuando el palo de triunfo es* `p` *(es decir, las cartas son del mismo palo y el valor de* `c1` *es mayor que el de* `c2` *o* `c1` *es del palo de triunfo). Por ejemplo,*

```
ganaCarta Corazones (Carta Sota Picas) (Carta (Numérico 5) Picas)
⇝ True
ganaCarta Corazones (Carta (Numérico 3) Picas) (Carta Sota Picas)
⇝ False
ganaCarta Corazones (Carta (Numérico 3) Corazones) (Carta Sota Picas)
⇝ True
ganaCarta Tréboles (Carta (Numérico 3) Corazones) (Carta Sota Picas)
⇝ False
```

Solución:

```
ganaCarta :: Palo -> Carta -> Carta -> Bool
ganaCarta triunfo c c'
    | palo c == palo c' = mayor (valor c) (valor c')
    | palo c == triunfo = True
    | otherwise         = False
```

Ejercicio 6.1.21. *Comprobar con QuickCheck si dadas dos cartas, una siempre gana a la otra.*

Solución: La propiedad es

```
prop_GanaCarta t c1 c2 =
    ganaCarta t c1 c2 || ganaCarta t c2 c1
```

La comprobación es

```
Main> quickCheck prop_GanaCarta
Falsifiable, after 0 tests:
Diamantes
Carta Rey Corazones
Carta As Tréboles
```

que indica que la propiedad no se verifica ya que cuando el triunfo es diamantes, ni el rey de corazones le gana al as de tréboles ni el as de tréboles le gana al rey de corazones.

Ejercicio 6.1.22. *Definir el tipo de datos* `Mano` *para representar una mano en el juego de cartas. Una mano es vacía o se obtiene añadiendo una carta a una mano. Hacer* `Mano` *instancia de* `Eq` *y* `Show`*.*

Solución:

```
data Mano = Vacía | Añade Carta Mano
            deriving (Eq, Show)
```

Ejercicio 6.1.23 (Avanzado)**.** *Definir un generador de manos para QuickCheck.*

Solución:

```
instance Arbitrary Mano where
    arbitrary =
        do cs <- arbitrary
           let mano []     = Vacía
               mano (c:cs) = Añade c (mano cs)
           return (mano cs)
```

Ejercicio 6.1.24. *Una mano gana a una carta* `c` *si alguna carta de la mano le gana a* `c`*. Definir la función*

```
ganaMano :: Palo -> Mano -> Carta -> Bool
```

tal que `(gana t m c)` *se verifica si la mano* `m` *le gana a la carta* `c` *cuando el triunfo es* `t`*. Por ejemplo,*

```
ganaMano Picas (Añade (Carta Sota Picas) Vacía) (Carta Rey Corazones)
⇝ True
ganaMano Picas (Añade (Carta Sota Picas) Vacía) (Carta Rey Picas)
⇝ False
```

Solución:

```
ganaMano :: Palo -> Mano -> Carta -> Bool
ganaMano triunfo Vacía     c'   = False
ganaMano triunfo (Añade c m) c' = ganaCarta triunfo c c' ||
                                  ganaMano triunfo m c'
```

Ejercicio 6.1.25. *Definir la función*

```
eligeCarta :: Palo -> Carta -> Mano -> Carta
```

tal que (`eligeCarta t c1 m`) *es la mejor carta de la mano* `m` *frente a la carta* `c` *cuando el triunfo es* `t`*. La estrategia para elegir la mejor carta es*

1. *Si la mano sólo tiene una carta, se elige dicha carta.*
2. *Si la primera carta de la mano es del palo de* `c1` *y la mejor del resto no es del palo de* `c1`*, se elige la primera de la mano,*
3. *Si la primera carta de la mano no es del palo de* `c1` *y la mejor del resto es del palo de* `c1`*, se elige la mejor del resto.*
4. *Si la primera carta de la mano le gana a* `c1` *y la mejor del resto no le gana a* `c1`*, se elige la primera de la mano,*
5. *Si la mejor del resto le gana a* `c1` *y la primera carta de la mano no le gana a* `c1`*, se elige la mejor del resto.*
6. *Si el valor de la primera carta es mayor que el de la mejor del resto, se elige la mejor del resto.*
7. *Si el valor de la primera carta no es mayor que el de la mejor del resto, se elige la primera carta.*

Solución:

```
eligeCarta :: Palo -> Carta -> Mano -> Carta
eligeCarta triunfo c1 (Añade c Vacía) = c                          -- 1
eligeCarta triunfo c1 (Añade c resto)
  | palo c == palo c1 && palo c' /= palo c1                 = c  -- 2
  | palo c /= palo c1 && palo c' == palo c1                 = c' -- 3
```

```
  | ganaCarta triunfo c  c1 && not (ganaCarta triunfo c' c1) = c  -- 4
  | ganaCarta triunfo c' c1 && not (ganaCarta triunfo c c1)  = c' -- 5
  | mayor (valor c) (valor c')                               = c' -- 6
  | otherwise                                                = c  -- 7
 where
  c' = eligeCarta triunfo c1 resto
```

Ejercicio 6.1.26. *Comprobar con QuickCheck que si una mano es ganadora, entonces la carta elegida es ganadora.*

Solución: La propiedad es

```
prop_eligeCartaGanaSiEsPosible triunfo c m =
    m /= Vacía ==>
    ganaMano triunfo m c == ganaCarta triunfo (eligeCarta triunfo c m) c
```

La comprobación es

```
Main> quickCheck prop_eligeCartaGanaSiEsPosible
Falsifiable, after 12 tests:
Corazones
Carta Rey Tréboles
Añade (Carta (Numérico 6) Diamantes)
       (Añade (Carta Sota Picas)
        (Añade (Carta Rey Corazones)
         (Añade (Carta (Numérico 10) Tréboles)
          Vacía)))
```

La carta elegida es el 10 de tréboles (porque tiene que ser del mismo palo), aunque el mano hay una carta (el rey de corazones) que gana.

6.2. Simplificación de definiciones

Ejercicio 6.2.1. *Simplifica la siguiente definición*

```
esGrande :: Integer -> Bool
esGrande n | n > 9999  = True
           | otherwise = False
```

Solución:

```
esGrande :: Integer -> Bool
esGrande n = n > 9999
```

Ejercicio 6.2.2. *Simplifica la siguiente definición*

```
resultadoEsGrande :: Integer -> Bool
resultadoEsGrande n = esGrande (f n) == True
```

Solución:

```
resultadoEsGrande :: Integer -> Bool
resultadoEsGrande n = esGrande (f n)
```

Ejercicio 6.2.3. *Simplifica la siguiente definición*

```
resultadoEsPequeño :: Integer -> Bool
resultadoEsPequeño n = esGrande (f n) == False
```

Solución:

```
resultadoEsPequeño :: Integer -> Bool
resultadoEsPequeño n = not (esGrande (f n))
```

6.3. Definición del tipo lista

Ejercicio 6.3.1. *Definir el tipo de datos* `Lista` *a partir de* `Vacía` *(para representar la lista vacía) y* `Añade` *(para representar la operación de añadir un elemento a una lista). Hacer* `Lista` *instancia de* `Show` *y* `Eq`*.*

Solución:

```
data Lista a = Vacía | Añade a (Lista a)
               deriving (Show, Eq)
```

Ejercicio 6.3.2. *Definir el tipo de datos* `Mano'` *para representar una mano en el juego de cartas usando* `Lista`*.*

Solución:

```
data Mano' = Lista Carta
```

Ejercicio 6.3.3. *Definir la función*

```
esVacía :: Lista a -> Bool
```

tal que `(esVacía l)` *se verifica si la lista* `l` *es vacía. Por ejemplo,*

```
esVacía Vacía            ⇝  True
esVacía (Añade 2 Vacía)  ⇝  False
```

Solución:

```
esVacía :: Lista a -> Bool
esVacía Vacía           = True
esVacía (Añade x lista) = False
```

Ejercicio 6.3.4. *Definir la función*

```
primero :: Lista a -> a
```

tal que `(primero l)` *es el primero de la lista* `l`*. Por ejemplo,*

```
primero (Añade 2 (Añade 5 Vacía))  ⇝  2
```

Solución:

```
primero :: Lista a -> a
primero (Añade x lista) = x
```

Se puede también definir para que muestre un error si la lista es vacía. Por ejemplo,

```
Main> primero' Vacía
Program error: la lista es vacia
```

```
primero' :: Lista a -> a
primero' Vacía           = error "la lista es vacia"
primero' (Añade x lista) = x
```

Ejercicio 6.3.5. *Definir la función*

```
ultimo :: Lista a -> a
```

tal que `(ultimo l)` *es el último elemento de la lista* `l`*. Por ejemplo,*

```
ultimo (Añade 2 (Añade 5 Vacía))  ⇝  5
```

Solución:

```
ultimo :: Lista a -> a
ultimo (Añade x Vacía) = x
ultimo (Añade x lista) = ultimo lista
```

Ejercicio 6.3.6. *Definir, usando la notación usual de listas, la función*

```
esVacía2 :: [a] -> Bool
```

tal que `(esVacía2 l)` *se verifica si la lista* `l` *es vacía. Por ejemplo,*

```
esVacía2 []   ⇝  True
esVacía2 [x]  ⇝  False
```

Solución:

```
esVacía2 :: [a] -> Bool
esVacía2 []       = True
esVacía2 (x:lista) = False
```

Ejercicio 6.3.7. *Definir, usando la notación usual de listas, la función*

```
primero2 :: [a] -> a
```

tal que `(primero2 l)` *es el primero de la lista* `l`*. Por ejemplo,*

```
primero2 [2,5]  ⇝  2
```

Solución:

```
primero2 :: [a] -> a
primero2 (x:lista) = x
```

Ejercicio 6.3.8. *Definir, usando la notación usual de listas, la función*

```
ultimo2 :: [a] -> a
```

tal que `(ultimo2 l)` *es el último elemento de la lista* `l`*. Por ejemplo,*

```
ultimo2 [2,5]  ⇝  5
```

Solución:

```
ultimo2 :: [a] -> a
ultimo2 [x]       = x
ultimo2 (x:lista) = ultimo2 lista
```

Ejercicio 6.3.9. *Definir la función*

```
suma :: Num a => [a] -> a
```

tal que `(suma xs)` *es la suma de los elementos de* `xs`*. Por ejemplo,*

```
suma [2,3,5]    ⇝  10
suma [2,3.4,5]  ⇝  10.4
```

Solución:

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

6.4. Concatenación de dos listas

Ejercicio 6.4.1. *Definir la función*

```
conc :: [a] -> [a] -> [a]
```

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 la predefinida* `(++)`*.*

Solución:

```
conc []     ys  = ys
conc (x:xs) ys  = x : (conc xs ys)
```

Ejercicio 6.4.2. *Detallar el cálculo de* `conc [2,3] [3,2,4,1]`

Solución:

```
conc [2,3] [3,2,4,1] = 2 : (conc [3] [3,2,4,1])
                     = 2 : (3 : (conc [] [3,2,4,1]))
                     = 2 : (3 : [3,2,4,1])
                     = 2 : [3,3,2,4,1]
                     = [2,3,3,2,4,1]
```

6.5. Inversa de una lista

Ejercicio 6.5.1. *Definir la función*

```
inversa :: [a] -> [a]
```

tal que `(inversa xs)` *es la inversa de la lista* `xs`*. Por ejemplo,*

```
inversa [1,3,2]  ⇝  [2,3,1]
```

Nota: `inversa` *es equivalente a la predefinida* `reverse`*.*

Solución:

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

Ejercicio 6.5.2. *Comprobar con QuickCheck que la inversa de la lista vacía es la lista vacía.*

Solución: La propiedad es

```
prop_InversaVacía :: Bool
prop_InversaVacía =
    inversa [] == ([] :: [Integer])
```

La comprobación es

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

Ejercicio 6.5.3. *Comprobar con QuickCheck que la inversa de una lista unitaria es la propia lista.*

Solución: La propiedad es

```
prop_InversaUnitaria :: Integer -> Bool
prop_InversaUnitaria x =
    inversa [x] == [x]
```

La comprobación es

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

Ejercicio 6.5.4. *Comprobar con QuickCheck que la inversa de la concatenación de* `xs` *e* `ys` *es la concatenación de la inversa de* `ys` *y la inversa de* `xs`*.*

Solución: La propiedad es

```
prop_InversaConcatenación :: [Integer] -> [Integer] -> Bool
prop_InversaConcatenación xs ys =
    inversa (xs ++ ys) == inversa ys ++ inversa xs
```

La comprobación es

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

Capítulo 7

Recursión y tipos de datos

Contenido

Estos ejercicios corresponden a la primera relación de ejercicios. Su objetivo es definir y razonar sobre funciones recursivas y tipos de datos.

7.1. La función máximo

Ejercicio 7.1.1. *Definir la función*

```
maxI :: Integer -> Integer -> Integer
```

tal que `(maxI x y)` *es el máximo de los números enteros* `x` *e* `y`*. Por ejemplo,*

```
maxI 2 5 ⇝ 5
maxI 7 5 ⇝ 7
```

Solución: La definición de `maxI` es

```
maxI :: Integer -> Integer -> Integer
maxI x y | x >= y    = x
         | otherwise = y
```

Ejercicio 7.1.2. *Verificar con QuickCheck que el máximo de* `x` *e* `y` *es mayor o igual que* `x` *y que* `y`*.*

Solución:

```
prop_MaxIMayor x y =
    maxI x y >= x &&  maxI x y >= y
```

La comprobación es

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

Ejercicio 7.1.3. *Verificar con QuickCheck que el máximo de* `x` *e* `y` *es* `x` *ó* `y`*.*

Solución:

```
prop_MaxIAlguno x y =
    maxI x y == x || maxI x y == y
```

La comprobación es

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

Ejercicio 7.1.4. *Verificar con QuickCheck que si* `x` *es mayor o igual que* `y`*, entonces el máximo de* `x` *e* `y` *es* `x`*.*

Solución:

```
prop_MaxIX x y =
    x >= y ==> maxI x y == x
```

La comprobación es

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

Ejercicio 7.1.5. *Verificar con QuickCheck que si* `y` *es mayor o igual que* `x`*, entonces el máximo de* `x` *e* `y` *es* `y`*.*

Solución:

```
prop_MaxIY x y =
    y >= x ==>  maxI x y == y
```

La comprobación es

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

7.2. Suma de cuadrados

Ejercicio 7.2.1. *Definir por recursión la función*

```
sumaCuadrados :: Integer -> Integer
```

tal que `(sumaCuadrados n)` *es la suma de los cuadrados de los números de* `1` *a* `n`*; es decir* $1^2 + 2^2 + 3^2 + \cdots + n^2$*. Por ejemplo,*

```
sumaCuadrados 4 ⇝ 30
```

Solución: La definición de `sumaCuadrados` es

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

Ejercicio 7.2.2. *Comprobar con QuickCheck si* `sumaCuadrados n` *es igual a* $\frac{n(n+1)(2n+1)}{6}$.

Solución: La propiedad es

```
prop_SumaCuadrados n =
    n >= 0 ==>
      sumaCuadrados n == n * (n+1) * (2*n+1) `div` 6
```

La comprobación es

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

Nótese que la condición `n >= 0` es necesaria. Si se la quitamos la propiedad

```
prop_SumaCuadrados2 n =
    sumaCuadrados n == n * (n+1) * (2*n+1) `div` 6
```

no se verifica

```
Main> quickCheck prop_SumaCuadrados2
Program error: pattern match failure: sumaCuadrados (-1)
```

7.3. Potencia

Ejercicio 7.3.1. *Definir la función*

```
potencia :: Integer -> Integer -> Integer
```

tal que `(potencia x n)` *es* x^n *donde* `x` *es un número entero y* `n` *es un número natural. Por ejemplo,*

```
potencia 2 3 ⇝ 8
```

Solución: La definición de `potencia` es

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

Ejercicio 7.3.2. *Definir la función*

```
potencia2 :: Integer -> Integer -> Integer
```

tal que `(potencia x n)` *es* x^n *usando la siguiente propiedad:*

$$x^n = \begin{cases} (x^2)^{n/2} & \text{si } n \text{ es par,} \\ x \times (x^2)^{(n-1)/2} & \text{si } n \text{ es impar} \end{cases}$$

Solución: La definición de `potencia2` es

```
potencia2 :: Integer -> Integer -> Integer
potencia2 x 0                 = 1
potencia2 x n | n>0 && even n = potencia2 (x*x) (n `div` 2)
              | n>0 && odd n  = x * potencia2 (x*x) ((n-1) `div` 2)
```

Ejercicio 7.3.3. *Comparar la eficiencia de las dos definiciones calculando* 3^{1000}.

Solución: La comparación es

```
Main> :set +s
Main> potencia 3 1000
1322070819480806636890455259752l...
(21033 reductions, 91721 cells)
Main> potencia2 3 1000
1322070819480806636890455259752l...
(1133 reductions, 2970 cells)
Main> :set -s
```

Ejercicio 7.3.4. *Comprobar con QuickCheck que las dos definiciones son equivalentes.*

Solución: La propiedad es

```
prop_Potencias x n =
    n >= 0 ==> potencia x n == potencia2 x n
```

La comprobación es

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

Ejercicio 7.3.5. *Definir la función*

```
llamadasPotencia :: Integer -> Integer -> Integer
```

tal que `(llamadasPotencia x)` *es el número de llamadas a la función* `potencia` *para calcular* `(potencia x)`. *Por ejemplo,*

```
llamadasPotencia 3 6 ⇝ 7
llamadasPotencia 3 7 ⇝ 8
llamadasPotencia 3 8 ⇝ 9
```

Solución: La definición de `llamadasPotencia` es

```
llamadasPotencia :: Integer -> Integer -> Integer
llamadasPotencia x 0         = 1
llamadasPotencia x n | n > 0 = 1 + llamadasPotencia x (n-1)
```

Ejercicio 7.3.6. *Definir la función*

```
llamadasPotencia2 :: Integer -> Integer -> Integer
```

tal que `(llamadasPotencia2 x)` *es el número de llamadas a la función* `potencia2` *para calcular* `(potencia2 x)`. *Por ejemplo,*

```
llamadasPotencia2 3 6 ⇝ 4
llamadasPotencia2 3 7 ⇝ 4
llamadasPotencia2 3 8 ⇝ 5
```

Solución: La definición de `llamadasPotencia2` es

```
llamadasPotencia2 :: Integer -> Integer -> Integer
llamadasPotencia2 x 0  = 1
llamadasPotencia2 x n
    | n>0 && even n    = 1 + llamadasPotencia2 x (n `div` 2)
    | n>0 && odd n     = 1 + llamadasPotencia2 x ((n-1) `div` 2)
```

Ejercicio 7.3.7. *Comparar el número de llamadas al calcular* 3^{1000}.

Solución: La comparación es

```
Main> llamadasPotencia 3 1000
1001
Main> llamadasPotencia2 3 1000
11
```

7.4. Las torres de Hanoi

Las torres de Hanoi es un rompecabeza que consta de tres postes que llamaremos A, B y C. Hay N discos de distintos tamaños en el poste A, de forma que no hay un disco situado sobre otro de menor tamaño. Los postes B y C están vacíos. Sólo puede moverse un disco a la vez y todos los discos deben de estar ensartados en algún poste. Ningún disco puede situarse sobre otro de menor tamaño. El problema consiste en colocar los N discos en algunos de los otros dos postes.

Ejercicio 7.4.1. *Diseñar una estrategia recursiva para resolver el problema de las torres de Hanoi.*

Solución: La estrategia recursiva es la siguiente:

- Caso base (N=1): Se mueve el disco de A a C.
- Caso inductivo (N=M+1): Se mueven M discos de A a C. Se mueve el disco de A a B. Se mueven M discos de C a B.

Ejercicio 7.4.2. *Definir la función*

```
hanoi :: Integer -> Integer
```

tal que `(hanoi n)` *es el número de movimientos necesarios para resolver el problema si inicialmente hay* `n` *discos en el poste A.*

Solución: La definición de `hanoi` es

```
hanoi :: Integer -> Integer
hanoi 1     = 1
hanoi (n+1) = 1+2*(hanoi n)
```

Ejercicio 7.4.3. *Calcular el número de movimientos necesarios si inicialmente hay 32 discos.*

Solución: El cálculo es

```
Main> hanoi 32
4294967295
```

7.5. Los números de Fibonacci

Los números de Fibonacci se definen por

$$\begin{aligned} F_0 &= 1 \\ F_1 &= 1 \\ F_{n+2} &= F_{n+1} + F_n \end{aligned}$$

por tanto, la sucesión de números de Fibonacci es $1, 1, 2, 3, 5, 8 \ldots$.

Ejercicio 7.5.1. *Definir, usando la anterior descripción, la función*

```
fib :: Integer -> Integer
```

tal que `(fib n)` *es el* `n`*-ésimo número de Fibonacci. Por ejemplo,*

```
fib 4 ⇝ 5
```

Solución: La definición de `fib` es

```
fib :: Integer -> Integer
fib 0     = 1
fib 1     = 1
fib (n+2) = fib (n+1) + fib n
```

Ejercicio 7.5.2. *Calcular los términos 10, 15, 20, 25 y 30 de la sucesión de Fibonacci con las estadísticas activadas. ¿Qué se observa?*

Solución: Los cálculos son

```
Main> :set +s
Main> fib 10
89
(4585 reductions, 6882 cells)
Main> fib 15
987
(51019 reductions, 76615 cells)
Main> fib 20
10946
(565981 reductions, 849959 cells)
Main> fib 25
121393
(6276997 reductions, 9426457 cells, 9 garbage collections)
Main> fib 30
1346269
(69613135 reductions, 104541274 cells, 105 garbage collections)
```

Se observa que se tiene una gran complejidad en los cálculos. En realidad, la complejidad de `fib` es $O(fib(n))$.

Ejercicio 7.5.3. *Existe una definición más eficiente para calcular los números de Fibonacci. Supongamos definida una función* `fibAux` *que satisface la siguiente propiedad*

```
fibAux i (fib j) (fib (j+1)) == fib (j+i)
```

(Nótese que esto no es una definición, sino una propiedad). Definir, usando la anterior propiedad, la función

```
fib2 :: Integer -> Integer
```

tal que `(fib2 n)` *es el* `n`*–ésimo número de Fibonacci. (Indicación: Intente sustituir* `i` *por* `n` *y* `j` *por 0 en la propiedad).*

Solución: La definición de `fib2` es

```
fib2 :: Integer -> Integer
fib2 n = fibAux n 1 1
```

Ejercicio 7.5.4. *Definir la función*

```
fibAux :: Integer -> Integer -> Integer -> Integer
```

para que se verifique la propiedad.

Solución: La definición de `fibAux` es

```
fibAux :: Integer -> Integer -> Integer -> Integer
fibAux 0     a b = a
fibAux (i+1) a b = fibAux i b (a+b)
```

Ejercicio 7.5.5. *Comprobar con QuickCheck que* `fib2` *verifica la propiedad.*

Solución: La propiedad es

```
prop_FibAux i j =
    i >= 0 && j >= 0 ==>
      fibAux i (fib2 j) (fib2 (j+1)) == fib2 (j+i)
```

y su comprobación es

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

Ejercicio 7.5.6. *Comprobar con QuickCheck que* `fib2` *y* `fib` *son equivalentes.*

Solución: La propiedad (limitada a 20 por cuestiones de eficiencia) es

```
prop_Fib2EquivFib n =
    n >= 0 && n <= 20 ==>
      fib n == fib2 n
```

y su comprobación es

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

Ejercicio 7.5.7. *Usando* `fib2`*, calcular los términos 10, 15, 20, 25 y 30 de la sucesión de Fibonacci con las estadísticas activadas. ¿Qué se observa?*

Solución: Los cálculos son

```
Main> :set +s
Main> fib2 10
89
(322 reductions, 487 cells)
Main> fib2 15
987
(467 reductions, 708 cells)
Main> fib2 20
10946
(612 reductions, 931 cells)
Main> fib2 25
121393
(757 reductions, 1156 cells)
Main> fib2 30
1346269
(902 reductions, 1382 cells)
Main>
```

Se observa que la complejidad se ha reducido a lineal.

Ejercicio 7.5.8. *Calcular manualmente* `fib2 4`

Solución: El cálculo es

```
fib2 4 == fibAux 4 1 1
       == fibAux 3 1 2
       == fibAux 2 2 3
       == fibAux 1 3 5
       == fibAux 0 5 8
       == 5
```

7.6. Divisores

Ejercicio 7.6.1. *Definir la función*

```
divide :: Integer -> Integer -> Bool
```

tal que `(divide a b)` *se verifica si* `a` *divide a* `b`*. Por ejemplo,*

```
divide 2 10  ⇝  True
divide 4 10  ⇝  False
```

Solución:

```
divide :: Integer -> Integer -> Bool
divide a b = b `mod` a == 0
```

Ejercicio 7.6.2. *Definir la función*

```
siguienteDivisor :: Integer -> Integer -> Integer
```

tal que `(siguienteDivisor k n)` *es el menor número mayor o igual que* `k` *que divide a* `n`*. Por ejemplo,*

```
siguienteDivisor 30 24  ⇝  24
siguienteDivisor  6 24  ⇝  6
siguienteDivisor  9 24  ⇝  12
```

Solución:

```
siguienteDivisor :: Integer -> Integer -> Integer
siguienteDivisor k n
    | k >= n     = n
    | divide k n = k
    | otherwise  = siguienteDivisor (k+1) n
```

Ejercicio 7.6.3. *Comprobar con QuickCheck que si* `k` *es mayor que 0 y menor o igual que* `n`*, entonces* `(siguienteDivisor k n)` *está entre* `k` *y* `n`*.*

Solución: La propiedad es

```
prop_SiguienteDivisor k n =
    0 < k && k <= n ==> k <= m && m <= n
    where m = siguienteDivisor k n
```

La comprobación es

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

Ejercicio 7.6.4. *Comprobar con QuickCheck que si* `k` *es mayor que 0 y menor o igual que* `n`*, entonces* `(siguienteDivisor k n)` *es un divisor de* `n`*,*

Solución: La propiedad es

```
prop_SiguienteDivisorEsDivisor k n =
    0 < k && k <= n ==> divide (siguienteDivisor k n)  n
```

La comprobación es

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

Ejercicio 7.6.5. *Definir la función*

```
menorDivisor :: Integer -> Integer
```

tal que `(menorDivisor n)` *es el menor divisor de* `n` *mayor que 1. Por ejemplo,*

```
menorDivisor 15  ⇝  3
menorDivisor 17  ⇝  17
```

Solución:

```
menorDivisor :: Integer -> Integer
menorDivisor n = siguienteDivisor 2 n
```

Ejercicio 7.6.6. *Comprobar con QuickCheck que si* `n` *es mayor que cero, entonces el menor divisor de* `n` *está entre 1 y* `n`*.*

Solución: La propiedad es

```
prop_MenorDivisor n =
    n > 0 ==> 1 <= m && m <= n
    where m = menorDivisor n
```

La comprobación es

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

Ejercicio 7.6.7. *Comprobar con QuickCheck que si* `n` *es mayor que cero, entonces el menor divisor de* `n` *divide a* `n`*.*

Solución: La propiedad es

```
prop_MenorDivisorEsDivisor n =
    n > 0 ==> divide (menorDivisor n) n
```

La comprobación es

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

Ejercicio 7.6.8. *Comprobar con QuickCheck que si* `n` *es mayor que cero, entonces el menor divisor de* `n` *es menor o igual que cualquier otro divisor de* `n` *que sea mayor que 1.*

Solución:

```
prop_MenorDivisorEsMenor n m =
    n > 0 ==>
    2 <= m && m <= n && divide m n ==> (menorDivisor n) <= m
```

La comprobación es

```
Main> quickCheck prop_MenorDivisorEsMenor
Arguments exhausted after 27 tests.
```

Ejercicio 7.6.9. *Definir la función*

```
númeroDivisoresDesde :: Integer -> Integer -> Integer
```

tal que `(númeroDivisoresDesde k n)` *es el número de divisores de* `n` *que son mayores o iguales que* `k`*. Por ejemplo,*

```
númeroDivisoresDesde 10 24  ⇝  2
```

Solución:

```
númeroDivisoresDesde :: Integer -> Integer -> Integer
númeroDivisoresDesde k n
    | k >= n     = 1
    | divide k n = 1 + númeroDivisoresDesde (siguienteDivisor (k+1) n) n
    | otherwise  = númeroDivisoresDesde (siguienteDivisor (k+1) n) n
```

Ejercicio 7.6.10. *Definir la función*

```
númeroDivisores :: Integer -> Integer
```

tal que `(númeroDivisores n)` *es el número de divisores de* `n`*. Por ejemplo,*

```
numeroDivisores 24  ⇝  8
numeroDivisores  7  ⇝  2
```

Solución:

```
númeroDivisores :: Integer -> Integer
númeroDivisores n = númeroDivisoresDesde 1 n
```

Ejercicio 7.6.11. *Comprobar con QuickCheck que si* `n` *es mayor que 1 y el menor divisor de* `n` *mayor que 1 es* `n`*, entonces* `n` *tiene exactamente 2 divisores (es decir,* `n` *es primo).*

Solución: La propiedad es

```
prop_Primo n =
    n > 1 && menorDivisor n == n ==> númeroDivisores n == 2
```

La comprobación es

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

7.7. Multiplicación de una lista de números

Ejercicio 7.7. *Definir la función*

```
multiplica :: Num a => [a] -> a
```

tal que `(multiplica xs)` *es la multiplicación de los números de la lista* `xs`*. Por ejemplo,*

```
multiplica [2,5,3]  ⇝  30
```

(Nota: La función `multiplica` *es quivalente a la predefinida* `product`*).*

Solución:

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

7.8. Eliminación de elementos duplicados

En muchas situaciones, las listas no deben de tener elementos repetidos. Por ejemplo, una baraja de cartas no debe de contener la misma carta dos veces.

Ejercicio 7.8.1. *Definir la función*

```
duplicados :: Eq a => [a] -> Bool
```

tal que `(duplicados xs)` *se verifica si la lista* `xs` *contiene elementos duplicados, Por ejemplo,*

```
duplicados [1,2,3,4,5] ⇝ False
duplicados [1,2,3,2]   ⇝ True
```

Solución: La definición es

```
duplicados :: Eq a => [a] -> Bool
duplicados []     = False
duplicados (x:xs) = elem x xs || duplicados xs
```

Ejercicio 7.8.2. *Definir la función*

```
eliminaDuplicados1 :: Eq a => [a] -> [a]
```

tal que `(eliminaDuplicados xs)` *es una lista que contiene los mismos elementos que* `xs` *pero sin duplicados. Por ejemplo,*

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

Solución: Presentamos dos definiciones. La primera definición es

```
eliminaDuplicados1 :: Eq a => [a] -> [a]
eliminaDuplicados1 []     = []
eliminaDuplicados1 (x:xs) = x : eliminaDuplicados1 (elimina x xs)
```

donde `elimina x xs` es la lista obtenida al eliminar todas las ocurrencias del elemento `x` en la lista `xs`

```
elimina :: Eq a => a -> [a] -> [a]
elimina x []                 = []
elimina x (y:ys) | x == y    = elimina x ys
                 | otherwise = y : elimina x ys
```

La segunda definición es

```
eliminaDuplicados2 :: Eq a => [a] -> [a]
eliminaDuplicados2 []                  = []
eliminaDuplicados2 (x:xs) | elem x xs = eliminaDuplicados2 xs
                          | otherwise = x : eliminaDuplicados2 xs
```

Nótese que en la segunda definición el orden de los elementos del resultado no se corresponde con el original. Por ejemplo,

```
eliminaDuplicados2 [1,3,1,2,3,2,1] ⇝ [3,2,1]
```

Sin embargo, se verifica la siguiente propiedad que muestra que las dos definiciones devuelven el mismo conjunto

```
prop_EquivEliminaDuplicados :: [Int] -> Bool
prop_EquivEliminaDuplicados xs =
    (reverse . eliminaDuplicados2 . reverse) xs == eliminaDuplicados1 xs
```

En efecto,

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

En lo sucesivo usaremos como definición de `eliminaDuplicados` la primera

```
eliminaDuplicados :: Eq a => [a] -> [a]
eliminaDuplicados = eliminaDuplicados1
```

Ejercicio 7.8.3. *Comprobar con QuickCheck que siempre el valor de* `eliminaDuplicados` *es una lista sin duplicados.*

Solución: La propiedad es

```
prop_duplicadosEliminados :: [Int] -> Bool
prop_duplicadosEliminados xs = not (duplicados (eliminaDuplicados xs))
```

y la comprobación es

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

Ejercicio 7.8.4. *¿Se puede garantizar con la propiedad anterior que* `eliminaDuplicados` *se comporta correctamente? En caso negativo, ¿qué propiedad falta?*

Solución: La propiedad anterior no garantiza que `eliminaDuplicados` se comporta correctamente, ya que la función que siempre devuelve la lista vacía también verifica la propiedad pero no se comporta como deseamos.
Lo que falta es una propiedad que garantice que todos los elementos de la lista original ocurren en el resultado

```
prop_EliminaDuplicadosMantieneElementos :: [Int] -> Bool
prop_EliminaDuplicadosMantieneElementos xs =
    contenido xs (eliminaDuplicados xs)
```

donde `contenido xs ys` se verifica si todos los elementos de `xs` pertenecen a `ys`

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

7.9. Fechas

Ejercicio 7.9.1. *Definir el tipo de datos* `Mes` *para representar los doce meses y hacerlo instancia de* `Eq` *y* `Show`.

Solución:

```
data Mes = Enero
         | Febrero
         | Marzo
         | Abril
         | Mayo
         | Junio
         | Julio
         | Agosto
         | Septiembre
         | Octubre
         | Noviembre
         | Diciembre
         deriving (Eq, Show)
```

Ejercicio 7.9.2. *Definir la función*

```
divisible :: Int -> Int -> Bool
```

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
```

Ejercicio 7.9.3. *La definición de año bisiesto es*

- *un año divisible por 4 es un año bisiesto (por ejemplo 2008);*
- *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).*

Definir la función

```
bisiesto :: Int -> Bool
```

tal que `(bisiesto a)` *se verifica si el año* `a` *es bisiesto. Por ejemplo,*

```
bisiesto 2008  ⇝  True
bisiesto 1900  ⇝  False
bisiesto 2000  ⇝  True
bisiesto 2007  ⇝  False
```

Solución:

```
bisiesto :: Int -> Bool
bisiesto a =
    divisible a 4 && (not(divisible a 100) || divisible a 400)
```

Ejercicio 7.9.4. *Definir la función*

```
díasDelMes :: Mes -> Int -> Int
```

tal que `(díasDelMes m a)` *es el número de días del mes* `m` *del año* `a`*. Por ejemplo,*

```
díasDelMes Febrero 2008  ⇝  29
díasDelMes Febrero 2007  ⇝  28
```

Solución:

```
díasDelMes :: Mes -> Int -> Int
díasDelMes Enero a      = 31
díasDelMes Febrero a
    | bisiesto a        = 29
    | otherwise         = 28
díasDelMes Marzo      a = 31
díasDelMes Abril      a = 30
díasDelMes Mayo       a = 31
díasDelMes Junio      a = 30
díasDelMes Julio      a = 31
díasDelMes Agosto     a = 31
díasDelMes Septiembre a = 30
díasDelMes Octubre    a = 31
díasDelMes Noviembre  a = 30
díasDelMes Diciembre  a = 31
```

Ejercicio 7.9.5. *Definir el tipo* `Fecha` *para representar las fechas mediante el día, el mes y el año. Por ejemplo,*

```
Main> :t F 3 Enero 2000
F 3 Enero 2000 :: Fecha
Main> :i Fecha
-- type constructor
data Fecha

-- constructors:
F :: Int -> Mes -> Int -> Fecha
-- selectors:
día :: Fecha -> Int
mes :: Fecha -> Mes
año :: Fecha -> Int
```

Solución:

```
data Fecha = F {día::Int, mes::Mes, año::Int}
```

Ejercicio 7.9.6. *Definir la función*

```
fechaVálida :: Fecha -> Bool
```

tal que `(fechaVálida f)` *se verifica si* `f` *es una fecha válida. Por ejemplo,*

```
fechaVálida (F 29 Febrero 2008)  ⇝  True
fechaVálida (F  0 Febrero 2008)  ⇝  False
fechaVálida (F 29 Febrero 2007)  ⇝  False
```

Solución:

```
fechaVálida :: Fecha -> Bool
fechaVálida f = 1 <= día f &&
                día f <= díasDelMes (mes f) (año f)
```

Capítulo 8

Listas y comprensión

Contenido

Esta es la segunda relación de ejercicios correspondientes a la tercera semana. Su objetivo es practicar con listas y con definiciones por comprensión. Se usarán las siguientes funciones predefinidas:

- `or :: [Bool] -> Bool` tal que `(or xs)` se verifica si algún elemento de `xs` es verdadero.
- `and :: [Bool] -> Bool` tal que `(and xs)` se verifica si todos los elementos de `xs` son verdaderos.
- `nub :: Eq a => [a] -> [a]` tal que `(nub xs)` es la lista `xs` sin elementos duplicados. Para usar `nub` hay que escribir `import Data.List` al principio del fichero.

8.1. Reconocimiento de permutaciones

Ejercicio 8.1.1. *Una permutación de una lista es otra lista con los mismos elementos, pero posiblemente en distinto orden. Por ejemplo, [1,2,1] es una permutación de [2,1,1] pero no de*

[1,2,2]. Definir la función

```
esPermutación :: Eq a => [a] -> [a] -> Bool
```

tal que `(esPermutación xs ys)` *se verifique si* `xs` *es una permutación de* `ys`*. Por ejemplo,*

```
esPermutación [1,2,1] [2,1,1] ⇝ True
esPermutación [1,2,1] [1,2,2] ⇝ False
```

Solución: La definición es

```
esPermutación :: Eq a => [a] -> [a] -> Bool
esPermutación []     []     = True
esPermutación []     (y:ys) = False
esPermutación (x:xs) ys     = elem x ys && esPermutación xs (borra x ys)
```

donde `borra x xs` es la lista obtenida borrando una ocurrencia de `x` en la lista `xs`. Por ejemplo,

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

```
borra :: Eq a => a -> [a] -> [a]
borra x []                 = []
borra x (y:ys) | x == y    = ys
               | otherwise = y : borra x ys
```

(*Nota:* la función `borra` es la función `delete` de la librería `List`).

Ejercicio 8.1.2. *Comprobar con QuickCheck que si una lista es una permutación de otra, las dos tienen el mismo número de elementos.*

Solución: La propiedad es

```
prop_PemutaciónConservaLongitud :: [Int] -> [Int] -> Property
prop_PemutaciónConservaLongitud xs ys =
    esPermutación xs ys ==> length xs == length ys
```

y la comprobación es

```
Main> quickCheck prop_PemutaciónConservaLongitud
Arguments exhausted after 86 tests.
```

Ejercicio 8.1.3. *Comprobar con QuickCheck que la inversa de una lista es una permutación de la lista.*

Solución: La propiedad es

```
prop_InversaEsPermutación :: [Int] -> Bool
prop_InversaEsPermutación xs =
    esPermutación (reverse xs) xs
```

y la comprobación es

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

8.2. Ordenación por inserción

Ejercicio 8.2.1. *Definir la función*

```
ordenada :: Ord a => [a] -> Bool
```

tal que `(ordenada xs)` *se verifica si la lista* `xs` *está ordenada de menor a mayor. Por ejemplo,*

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

Solución:

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

Ejercicio 8.2.2. *Definir la función*

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

tal que `(inserta e xs)` *inserta el elemento* `e` *en la lista* `xs` *delante del primer elemento de* `xs` *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
```

Ejercicio 8.2.3. *Comprobar que al insertar un elemento en una lista ordenada se obtiene una lista ordenada.*

Solución: La propiedad es

```
prop_inserta :: Integer -> [Integer] -> Property
prop_inserta e xs =
    ordenada xs ==> ordenada (inserta e xs)
```

La comprobación es

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

Ejercicio 8.2.4. *Definir la función*

```
ordenaPorInserción :: Ord a => [a] -> [a]
```

tal que `(ordenaPorInserción xs)` *es la lista* `xs` *ordenada mediante inserción, Por ejemplo,*

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

Solución:

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

Ejercicio 8.2.5. *Escribir y comprobar con QuickCheck las propiedades que aseguran que ordenaPorInserción es una función de ordenación correcta.*

Solución: La primera propiedad es que `(ordenaPorInserción xs)` es una lista ordenada.

```
prop_Ordenada :: [Integer] -> Bool
prop_Ordenada xs =
    ordenada (ordenaPorInserción xs)
```

Su comprobación es

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

La segunda propiedad es que `(ordenaPorInserción xs)` es una permutación de `xs`.

```
prop_Permutación :: [Integer] -> Bool
prop_Permutación xs =
    esPermutación (ordenaPorInserción xs) xs
```

Su comprobación es

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

8.3. El triángulo de Pascal

El triángulo de Pascal es un triángulo de números

```
      1
     1 1
    1 2 1
  1  3 3  1
 1 4  6  4 1
1 5 10 10 5 1
.............
```

construido de la siguiente forma

- La primera fila está formada por el número 1;
- las filas siguientes se construyen sumando los números adyacentes de la fila superior y añadiendo un 1 al principio y al final de la fila.

Ejercicio 8.3.1. *Definir la función*

```
pascal :: Integer -> [Integer]
```

tal que `(pascal n)` *es la* `n`*–ésima fila del triángulo de Pascal. Por ejemplo,*

```
pascal 6 ⇝ [1,5,10,10,5,1]
```

Solución:

```
pascal :: Integer -> [Integer]
pascal 1 = [1]
pascal n = [1] ++ [ x+y | (x,y) <- pares (pascal (n-1)) ] ++ [1]
```

donde `pares xs` es la lista formada por los pares de elementos adyacentes de la lista `xs`. Por ejemplo,

```
pares [1,4,6,4,1] ⇝ [(1,4),(4,6),(6,4),(4,1)]
```

La definición de pares es

```
pares :: [a] -> [(a,a)]
pares (x:y:xs) = (x,y) : pares (y:xs)
pares _        = []
```

Otra definición de `pares`, usando `zip`, es

```
pares' :: [a] -> [(a,a)]
pares' xs = zip xs (tail xs)
```

Las definiciones son equivalentes

```
prop_ParesEquivPares' :: [Integer] -> Bool
prop_ParesEquivPares' xs =
    pares xs == pares' xs
```

Ejercicio 8.3.2. *Comprobar con QuickCheck, que la fila* `n`*–ésima del triángulo de Pascal tiene* `n` *elementos.*

Solución: La propiedad es

```
prop_Pascal :: Integer -> Property
prop_Pascal n =
    n >= 1 ==>
      fromIntegral (length (pascal n)) == n
```

Nótese el uso de la función `fromIntegral` para tansformar el valor de `length (pascal n)` de `Int` a `Integer`. La comprobación es

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

Ejercicio 8.3.3. *Comprobar con QuickCheck, que el* `m`*–ésimo elemento de la fila* `n+1`*–ésima del triángulo de Pascal es el número combinatorio* $\binom{n}{m} = \frac{n!}{k!\ (n-k)!}$.

Solución: La propiedad es

```
prop_Combinaciones :: Integer -> Property
prop_Combinaciones n =
    n >= 1 ==>
      pascal n == [comb (n-1) m | m <- [0..n-1]]
```

donde `fact n` es el factorial de `n`

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

y `comb n k` es el número combinatorio $\binom{n}{k}$.

```
comb :: Integer -> Integer -> Integer
comb n k = (fact n) `div` ((fact k) * (fact (n-k)))
```

La comprobación es

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

8.4. Cálculo de primos mediante la criba de Erastótenes

La criba de Erastótenes es un método para calcular números primos. Se comienza escribiendo todos los números desde 2 hasta (supongamos) 100. El primer número (el 2) es primo. Ahora eliminamos todos los múltiplos de 2. El primero de los números restantes (el 3) también es primo. Ahora eliminamos todos los múltiplos de 3. El primero de los números restantes (el 5) también es primo ... y así sucesivamente. Cuando no quedan números, se han encontrado todos los números primos en el rango fijado.

Ejercicio 8.4.1. *Definir la función*

```
elimina :: Int -> [Int] -> [Int]
```

tal que `(elimina n xs)` *es la lista obtenida elimando en la lista* `xs` *los múltiplos de* `n`*. Por ejemplo,*

```
elimina 3 [2,3,8,9,5,6,7]  ⇝  [2,8,5,7]
```

Solución:

```
elimina :: Int -> [Int] -> [Int]
elimina n xs = [ x | x <- xs, x `mod` n /= 0 ]
```

Ejercicio 8.4.2. *Definir la función*

```
criba :: [Int] -> [Int]
```

tal que `(criba xs)` *es la lista obtenida cribando la lista* `xs` *con el método descrito anteriormente. Por ejemplo,*

```
criba [2..20]  ⇝  [2,3,5,7,11,13,17,19]
```

Solución:

```
criba :: [Int] -> [Int]
criba []     = []
criba (n:ns) = n : criba (elimina n ns)
```

Ejercicio 8.4.3. *Definir la constante* `primos1a100` *cuyo valor es la lista de los números primos menores o iguales que 100.*

Solución: La definición es

```
primos1a100 :: [Int]
primos1a100 = criba [2..100]
```

El cálculo es

```
Main> primos1a100
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97]
```

Ejercicio 8.4.4. *Comprobar con QuickCheck que todos los elementos obtenidos con la criba son primos.*

Solución: La propiedad es

```
prop_Criba n =
  n >= 2 ==>
  and [ esPrimo x | x <- criba [2..n] ]
    where
      esPrimo n  = factores n == [1,n]
      factores n = [ k | k <- [1..n], n `mod` k == 0 ]
```

La comprobación es

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

8.5. Conjetura de Goldbach

Ejercicio 8.5.1. *Definir la función*

```
esPrimo100 :: Int -> Bool
```

tal que `(esPrimo100 n)` *se verifica si* `x` *es un número primo en el rango de 2 a 100. Por ejemplo,*

```
esPrimo100 17   ⇝  True
esPrimo100 27   ⇝  False
esPrimo100 101  ⇝  False
```

Solución:

```
esPrimo100 :: Int -> Bool
esPrimo100 n = n `elem` primos1a100
```

Ejercicio 8.5.2. *Definir la función*

```
esSuma2Primos100 :: Int -> Bool
```

tal que `(esSuma2Primos100 n)` *se verifica si* `x` *es la suma de dos primos en el rango de 2 a 100. Por ejemplo,*

```
esSuma2Primos100 26  ⇝  True
esSuma2Primos100 27  ⇝  False
```

Solución:

```
esSuma2Primos100 :: Int -> Bool
esSuma2Primos100 n =
  not (null [ (a,b)
            | a <- primos1a100
            , b <- primos1a100
            , n == a+b
            ])
```

Ejercicio 8.5.3. *La conjetura de Goldbach afirma que todo número par mayor que 2 puede expresarse como suma de dos primos. Por ejemplo, 4=2+2, 6=3+3, 8=3+5. Comprobar la conjetura para todos los números pares en el rango de 4 a 100.*

Solución: La conjetura se verifica si lo hace la siguiente propiedad

```
prop_Goldbach =
    null [ n | n <- [4..100], even n, not (esSuma2Primos100 n) ]
```

La comprobación es

```
Main> prop_Goldbach
True
```

8.6. Multiconjuntos

Ejercicio 8.6.1. *Definir por comprensión la función*

```
todosOcurrenEn :: Eq a => [a] -> [a] -> Bool
```

tal que `(todosOcurrenEn xs ys)` *se verifica si todos los elmentos de* `xs` *son elementos de* `ys`. *Por ejemplo,*

```
todosOcurrenEn [1,5,2,5] [5,1,2,4]  ⇝  True
todosOcurrenEn [1,5,2,5] [5,2,4]    ⇝  False
```

Solución:

```
todosOcurrenEn :: Eq a => [a] -> [a] -> Bool
todosOcurrenEn xs ys = and [ x `elem` ys | x <- xs ]
```

Ejercicio 8.6.2. *Definir por comprensión la función*

```
igualesElementos :: Eq a => [a] -> [a] -> Bool
```

tal que `(igualesElementos xs ys)` *se verifica si* `xs` *e* `ys` *tienen exactamente los mismos elementos. Por ejemplo,*

```
igualesElementos [1,5,2,5] [5,1,2]    ⇝  True
igualesElementos [1,5,2,5] [5,1,2,4]  ⇝  False
```

Solución:

```
igualesElementos :: Eq a => [a] -> [a] -> Bool
igualesElementos xs ys = todosOcurrenEn xs ys && todosOcurrenEn ys xs
```

Ejercicio 8.6.3. *Definir por comprensión la función*

```
númeroOcurrencias :: Eq a => a -> [a] -> Int
```

tal que `(númeroOcurrencias x ys)` *es el número de ocurrencias del elemento* `x` *en la lista* `ys`. *Por ejemplo,*

```
númeroOcurrencias 3 [1,3,2,3,5]     ⇝  2
númeroOcurrencias 'a' "Salamandra"  ⇝  4
```

Solución:

```
númeroOcurrencias :: Eq a => a -> [a] -> Int
númeroOcurrencias x xs = length [ x' | x' <- xs, x == x' ]
```

Ejercicio 8.6.4. *En cierta forma, las listas son semejantes a los conjuntos: ambas son colecciones de elementos. Pero el orden de los elementos en una lista importa, mientras que no importa en los conjuntos, y el número de ocurrencias en una lista importa, mientras que no importa en los conjuntos.*
El concepto de multiconjunto está entre el de lista y el de conjunto: el número de ocurrencias importa, pero no importa el orden de los elementos. Una manera de representar los multiconjuntos es una lista de pares de valores y el número de veces que el valor ocurre: por ejemplo, [('a',1), ('b',2)].
Definir la función

```
multiconjunto :: Eq a => [a] -> [(a,Int)]
```

tal que `(multiconjunto xs)` *es el multiconjunto correspondiente a la lista* `xs`*. Por ejemplo,*

```
multiconjunto [1,2,3,2,1,2] ⇝  [(1,2),(2,3),(3,1)]
multiconjunto "rareza"      ⇝  [('r',2),('a',2),('e',1),('z',1)]
```

Solución:

```
multiconjunto :: Eq a => [a] -> [(a,Int)]
multiconjunto xs = [ (x, númeroOcurrencias x xs) | x <- nub xs ]
```

Ejercicio 8.6.5. *Definir la función*

```
númeroDeElementos :: [(a,Int)] -> Int
```

tal que `(númeroDeElementos xs)` *es el número de elementos del multiconjunto* `xs`*. Por ejemplo,*

```
númeroDeElementos [(1,5),(2,4),(3,1)]  ⇝  10
```

Solución:

```
númeroDeElementos :: [(a,Int)] -> Int
númeroDeElementos xs = sum [ n | (x,n) <- xs ]
```

Ejercicio 8.6.6. *Comprobar con QuickCheck que si* `xs` *es una lista entonces* `(multiconjunto xs)` *tiene el mismo número de elementos que* `xs`*.*

Solución: La propiedad es

```
prop_multiconjunto1 :: [Int] -> Bool
prop_multiconjunto1 xs =
    númeroDeElementos (multiconjunto xs) == length xs
```

y la comprobación es

```
Main> quickCheck prop_multiconjunto1
OK, passed 100 tests.
```

Ejercicio 8.6.7. *Definir por comprensión la función*

```
lista :: Eq a => [(a,Int)] -> [a]
```

tal que `(lista xs)` *es la lista correspondiente al multiconjunto* `xs`*. Por ejemplo,*

```
lista [(1,2),(2,3),(3,1)]                ⇝  [1,1,2,2,2,3]
lista [('r',2),('a',2),('e',1),('z',1)]  ⇝  "rraaez"
```

Solución:

```
lista :: Eq a => [(a,Int)] -> [a]
lista xs = [ x | (x,n) <- xs, i <- [1..n] ]
```

Ejercicio 8.6.8. *Comprobar con QuickCheck que si* `xs` *es una lista entonces tienen los mismos elementos* `(lista (multiconjunto xs))` *y* `xs`*.*

Solución: La propiedad es

```
prop_multiconjunto2 :: [Int] -> Bool
prop_multiconjunto2 xs =
    igualesElementos (lista (multiconjunto xs)) xs
```

y la comprobación es

```
Main> quickCheck prop_multiconjunto2
OK, passed 100 tests.
```

8.7. Posiciones

Los elementos de una lista aparecen en una determinada posición. Por ejemplo, `a` aparece en `cama` en las posiciones 1 y 3 (nótese que se comienza a contar en 0).

Ejercicio 8.7.1. *Definir la función*

```
posiciones :: [a] -> [(a,Int)]
```

tal que `(posiciones xs)` *es la lista de pares formados por los elementos de* `xs` *y sus posiciones. Por ejemplo,*

```
posiciones "cama"  ⇝  [('c',0),('a',1),('m',2),('a',3)]
```

Indicación: Usar la predefinida `zip`.

Solución:

```
posiciones :: [a] -> [(a,Int)]
posiciones xs = zip xs [0..]
```

Ejercicio 8.7.2. *Definir la función*

```
primeraPosición :: Eq a => a -> [a] -> Int
```

tal que `(primeraPosición x xs)` *es la primera posición del elemento* `x` *en la lista* `xs`. *Por ejemplo,*

```
primeraPosición 'a' "cama"  ⇝  1
```

Solución:

```
primeraPosición :: Eq a => a -> [a] -> Int
primeraPosición x xs = head [ i | (x',i) <- posiciones xs, x' == x ]
```

Ejercicio 8.7.3. *Comprobar con QuickCheck que si* `x` *es un elemento de* `xs` *y* `n` *es la primera posición de* `x` *en* `xs`, *entonces el elemento de* `xs` *en la posición* `n` *es* `x`.

Solución: La propiedad es

```
prop_PrimeraPosiciónEsX :: Int -> [Int] -> Property
prop_PrimeraPosiciónEsX x xs =
    x `elem` xs ==>
      xs !! (primeraPosición x xs) == x
```

y la comprobación es

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

Ejercicio 8.7.4. *Comprobar con QuickCheck que si* `x` *es un elemento de* `xs` *y* `n` *es la primera posición de* `x` *en* `xs`, *entonces* `x` *no pertenece al segmento inicial de* `xs` *de longitud* `n`.

Solución: La propiedad es

```
prop_PrimeraPosiciónEsPrimera :: Int -> [Int] -> Property
prop_PrimeraPosiciónEsPrimera x xs =
    x `elem` xs ==>
      notElem x (take (primeraPosición x xs) xs)
```

y la comprobación es

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

Ejercicio 8.7.5. *Definir la función*

```
borra :: Eq a => Int -> a -> [a] -> [a]
```

tal que `(borra n x xs)` *es la lista obtenida borrando las primeras* `n` *ocurrencias de* `x` *en* `xs`*. Por ejemplo,*

```
borra 2 'a' "salamandra"  ⇝  "slmandra"
borra 7 'a' "salamandra"  ⇝  "slmndr"
```

Solución:

```
borra :: Eq a => Int -> a -> [a] -> [a]
borra 0 _ ys                 = ys
borra _ _ []                 = []
borra n x (y:ys) | x == y    = borra (n-1) x ys
                 | otherwise = y : borra n x ys
```

Ejercicio 8.7.6. *Comprobar con QuickCheck que el número de elementos borrados mediante* `(borra n x xs)` *es menor o igual que* `n`*.*

Solución: La propiedad es

```
prop_Borra :: Int -> Int -> [Int] -> Property
prop_Borra n x xs =
    n >= 0 ==>
      length (xs \\ borra n x xs) <= n
```

y la comprobación es

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

Ejercicio 8.7.7. *Comprobar con QuickCheck que todos los elementos borrados mediante* `(borra n x xs)` *son iguales a* `x`*.*

Solución: La propiedad es

```
prop_BorraSóloX :: Int -> Int -> [Int] -> Bool
prop_BorraSóloX n x xs =
  and [ x == x' | x' <- xs \\ borra n x xs ]
```

y la comprobación es

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

Ejercicio 8.7.8. *Definir la función*

```
borraPrimera :: Eq a => a -> [a] -> [a]
```

tal que `(borraPrimera x xs)` *es la lista obtenida borrando la primera ocurrencia de* `x` *en* `xs`. *Por ejemplo,*

```
borraPrimera 'a' "cama"  ⇝  "cma"
```

Solución:

```
borraPrimera :: Eq a => a -> [a] -> [a]
borraPrimera = borra 1
```

8.8. Ternas pitagóricas

Ejercicio 8.8.1. *Una terna pitagórica es una terna de números enteros* (a, b, c) *tal que* $a^2 + b^2 = c^2$. *Definir la función*

```
ternasPitagóricas :: Int -> [(Int,Int,Int)]
```

tal que `(ternasPitagóricas n)` *es la lista de las ternas pitagóricas* `(a,b,c)` *tales que* $1 \leq a \leq b \leq c \leq n$. *Por ejemplo,*

```
ternasPitagóricas 10  ⇝  [(3,4,5),(6,8,10)]
```

Solución:

```
ternasPitagóricas :: Int -> [(Int,Int,Int)]
ternasPitagóricas n = [ (a,b,c)
                      | a <- [1..n]
                      , b <- [a..n]
                      , c <- [b..n]
                      , a^2 + b^2 == c^2
                      ]
```

Otra definición más eficiente es

```
ternasPitagóricas' :: Int -> [(Int,Int,Int)]
ternasPitagóricas' n = [ (a,b,c)
                       | a <- [1..n]
                       , b <- [a..n]
                       , let c2 = a^2 + b^2
                             c  = floor (sqrt (fromIntegral c2))
                       , c <= n
                       , c^2 == c2
                       ]
```

Una comprobación de la mejora de la eficiencia es

```
Main> :set +s
Main> length (ternasPitagóricas 100)
52
(46955161 reductions, 72584119 cells, 75 garbage collections)
Main> length (ternasPitagóricas' 100)
52
(3186097 reductions, 4728843 cells, 4 garbage collections)
```

La propiedad que afirma que las dos definiciones son equivalentes es

```
prop_EquivTernasPitagóricas :: Int -> Bool
prop_EquivTernasPitagóricas n =
    ternasPitagóricas n == ternasPitagóricas' n
```

y su comprobación es

```
Main> quickCheck prop_EquivTernasPitagóricas
OK, passed 100 test
```

Capítulo 9

Funciones de entrada y salida. Generación de pruebas

Contenido

Esta es la tercera relación de ejercicios correspondientes a la cuarta semana. Su objetivo es practicar con las funciones de entrada y salida así como con la generación de datos para las pruebas con QuickCheck.

9.1. Copia de ficheros

Ejercicio 9.1.1. *Definir la función*

```
copiaFichero :: FilePath -> FilePath -> IO ()
```

tal que `(copiaFichero f1 f2)` *copia el fichero* `f1` *en el fichero* `f2`.

Solución:

```
copiaFichero :: FilePath -> FilePath -> IO ()
copiaFichero f1 f2 =
    do contenido <- readFile f1
       writeFile f2 contenido
```

9.2. Acción y escritura

Ejercicio 9.2.1. *Definir la función*

```
escribe :: Show a => IO a -> IO ()
```

tal que `(escribe io)` *ejecuta la acción* `io` *y escribe su resultado. Por ejemplo,*

```
Main> escribe (print "hola")
"hola"
()
```

```
escribe :: Show a => IO a -> IO ()
escribe io =
    do resultado <- io
       print resultado
```

Ejercicio 9.2.2. *Definir la función*

```
dosVeces :: Monad a => a b -> a (b,b)
```

que tal que `(dosVeces io)` *ejecuta dos veces la acción* `io`*. Por ejemplo,*

```
Main> escribe (dosVeces (print "hola"))
"hola"
"hola"
((),())
```

Solución:

```
dosVeces :: Monad a => a b -> a (b,b)
dosVeces io =
    do a <- io
       b <- io
       return (a,b)
```

9.3. Muestra de valores generados

Ejercicio 9.3.1. *Consultar la información sobre la clase Arbitrary de QuickCheck.*

Solución: La consulta es

```
Main> :i Arbitrary
-- type class
class Arbitrary a where
  arbitrary :: Gen a
  coarbitrary :: a -> Gen b -> Gen b

-- instances:
instance Arbitrary ()
instance Arbitrary Bool
instance Arbitrary Int
instance Arbitrary Integer
instance Arbitrary Float
instance Arbitrary Double
instance (Arbitrary a, Arbitrary b) => Arbitrary (a,b)
instance (Arbitrary a, Arbitrary b, Arbitrary c)
         => Arbitrary (a,b,c)
instance (Arbitrary a, Arbitrary b, Arbitrary c, Arbitrary d)
         => Arbitrary (a,b,c,d)
instance Arbitrary a => Arbitrary [a]
instance (Arbitrary a, Arbitrary b) => Arbitrary (a -> b)
```

Ejercicio 9.3.2. *Definir la función*

```
muestra :: Show a => Gen a -> IO ()
```

tal que `(muestra g)` *escribe 5 valores generados por el generador* `g`*. Por ejemplo,*

```
Main> muestra (arbitrary :: Gen Int)
0
0
-2
-2
3
Main> muestra (arbitrary :: Gen Int)
-1
-1
-2
```

```
-1
0
Main> muestra (arbitrary :: Gen Bool)
True
False
False
False
True
Main> muestra (arbitrary :: Gen Double)
1.5
0.0
4.66666666666667
0.8
4.0
Main> muestra (arbitrary :: Gen [Int])
[3,-3,-2,0,0]
[]
[0]
[1,2]
[]
Main> muestra (return True)
True
True
True
True
True
```

Solución:

```
muestra :: Show a => Gen a -> IO ()
muestra gen =
    sequence_
      [ do rnd <- newStdGen
           print (generate 5 rnd gen)
      | i <- [1..5]
      ]
```

Ejercicio 9.3.3. *Usar* `muestra` *para imprimir 5 pares de enteros.*

Solución: La generación es

```
Main> muestra (dosVeces (arbitrary :: Gen Integer))
(0,0)
```

```
(-2,0)
(5,-3)
(0,0)
(-2,3)
```

Ejercicio 9.3.4. *Definir el generador*

```
enteroPar :: Gen Integer
```

que genera números pares. Por ejemplo,

```
Main> muestra enteroPar
-8
8
0
10
-6
```

Solución:

```
enteroPar :: Gen Integer
enteroPar =
    do n <- arbitrary
       return (2*n)
```

Ejercicio 9.3.5. *Usando* `muestra` *y* `choose`*, imprimir 5 números entre 1 y 10.*

Solución: La generación es

```
Main> muestra (choose (1,10) :: Gen Integer)
10
5
1
3
10

Main> muestra (choose (1,10) :: Gen Integer)
10
10
2
7
7
```

Ejercicio 9.3.6. *Usando* `muestra` *y* `oneof`*, imprimir 5 veces un número de la lista [1,7].*

Solución: La generación es

```
Main> muestra (oneof [return 1, return 7])
7
1
1
7
7
```

9.4. Generación de listas

Ejercicio 9.4.1. *A veces se necesita generar listas de cierta longitud. Definir el generador*

```
listaDe :: Int -> Gen a -> Gen [a]
```

tal que `(listaDe n g)` *es una lista de* `n` *elementos, donde cada elemento es generado por* `g`*. Por ejemplo,*

```
Main> muestra (listaDe 3 (arbitrary :: Gen Int))
[-1,1,-1]
[-2,-4,-1]
[1,-1,0]
[1,-1,1]
[1,-1,1]
Main> muestra (listaDe 3 (arbitrary :: Gen Bool))
[False,True,False]
[True,True,False]
[False,False,True]
[False,False,True]
[True,False,True]
```

Solución:

```
listaDe :: Int -> Gen a -> Gen [a]
listaDe n g = sequence [ g | i <- [1..n] ]
```

Ejercicio 9.4.2. *Comprobar con QuickCheck que las listas generadas mediante* `(listaDe n _)` *son de longitud* `n`*.*

Solución: La propiedad es

```
prop_listaDe :: Int -> Property
prop_listaDe n =
    forAll (listaDe (abs n) (arbitrary :: Gen Int)) $ \xs ->
    length xs == (abs n)
```

y su verificación es

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

Ejercicio 9.4.3. *Definir el generador*

```
paresDeIgualLongitud :: Gen a -> Gen ([a],[a])
```

que genere pares de listas de igual longitud. Por ejemplo,

```
Main> muestra (paresDeIgualLongitud (arbitrary :: Gen Int))
([-4,5],[-4,2])
([],[])
([0,0],[-2,-3])
([2,-2],[-2,1])
([0],[-1])
Main> muestra (paresDeIgualLongitud (arbitrary :: Gen Bool))
([False,True,False],[True,True,True])
([True],[True])
([],[])
([False],[False])
([],[])
```

Solución:

```
paresDeIgualLongitud :: Gen a -> Gen ([a],[a])
paresDeIgualLongitud gen =
    do n <- arbitrary
       xs <- listaDe (abs n) gen
       ys <- listaDe (abs n) gen
       return (xs,ys)
```

Ejercicio 9.4.4. *Comprobar con QuickCheck que* `zip` *es la inversa de* `unzip`*; es decir, que si se descompone una lista de pares mediante* `unzip` *y se compone con* `zip` *se obtiene la lista original.*

Solución: La propiedad es

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

La comprobación es

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

Ejercicio 9.4.5. *Comprobar con QuickCheck que si se unen dos listas de igual longitud con* `zip` *y se separan con* `unzip` *se obtienen las listas originales. Usar* `collect` *para mostrar las longitudes de las listas usadas.*

Solución: La propiedad es

```
prop_UnzipZip :: [Int] -> [Int] -> Property
prop_UnzipZip xs ys =
    collect (length xs) $
    length xs == length ys ==>
    unzip (zip xs ys) == (xs,ys)
```

La comprobación es

```
Main> quickCheck prop_UnzipZip
OK, passed 100 tests.
53% 0.
15% 1.
5% 5.
5% 2.
4% 7.
4% 4.
3% 8.
2% 16.
2% 14.
1% 6.
1% 3.
1% 22.
1% 21.
1% 13.
1% 11.
1% 10.
```

Ejercicio 9.4.6. *Comprobar con QuickCheck que si se unen dos listas de igual longitud con* `zip` *y se separan con* `unzip` *se obtienen las listas originales. Usar* `paresDeIgualLongitud` *para generarlas y* `collect` *para mostrar las longitudes de las listas usadas.*

Solución: La propiedad es

```
prop_UnzipZip' :: Property
prop_UnzipZip' =
    forAll (paresDeIgualLongitud arbitrary) $ \(xs,ys) ->
    collect (length xs) $
    unzip (zip xs ys) == (xs :: [Int],ys :: [Int])
```

La comprobación es

```
Main> quickCheck prop_UnzipZip'
OK, passed 100 tests.
16% 1.
12% 0.
9% 9.
9% 3.
7% 6.
6% 5.
6% 13.
4% 8.
4% 7.
4% 2.
4% 14.
3% 4.
3% 16.
3% 15.
2% 12.
1% 38.
1% 29.
1% 22.
1% 21.
1% 19.
1% 18.
1% 11.
1% 10.
```

9.5. Mayorías parlamentarias

Ejercicio 9.5.1. *Definir el tipo de datos* `Partido` *para representar los partidos de un Parlamento. Los partidos son* `P1`, `P2`,..., `P8`. *La clase partido está contenida en* `Eq`, `Ord` *y* `Show`.

Solución:

```
data Partido
    = P1 | P2 | P3 | P4 | P5 | P6 | P7 | P8
    deriving ( Eq, Ord, Show )
```

Ejercicio 9.5.2. *Definir el tipo* `Escaños` *para representar el número de escaños que posee un partido en el parlamento.*

Solución:

```
type Escaños = Integer
```

Ejercicio 9.5.3. *Definir el tipo* `(Tabla a b)` *para representar una lista de pares de elementos el primero de tipo* `a` *y el segundo de tipo* `b`. *Definir* `Asamblea` *para representar una tabla de partidos y escaños.*

Solución:

```
type Tabla a b = [(a,b)]
type Asamblea  = Tabla Partido Escaños
```

Ejercicio 9.5.4. *Definir la función*

```
partidos :: Asamblea -> [Partido]
```

tal que `(partidos a)` *es la lista de partidos en la asamblea* `a`. *Por ejemplo,*

```
partidos [(P1,3),(P3,5),(P4,3)]  ==>  [P1,P3,P4]
```

Solución:

```
partidos :: Asamblea -> [Partido]
partidos a = [ p | (p,_) <- a ]
```

Ejercicio 9.5.5. *Definir la función*

```
escaños :: Asamblea -> Integer
```

tal que `(escaños a)` *es el número de escaños en la asamblea* `a`. *Por ejemplo,*

```
escaños [(P1,3),(P3,5),(P4,3)]  ==>  11
```

Solución:

```
escaños :: Asamblea -> Integer
escaños a = sum [ e | (_,e) <- a ]
```

Ejercicio 9.5.6. *Definir la función*

```
busca :: Eq a => a -> Tabla a b -> b
```

tal que `(busca x t)` *es el valor correspondiente a* `x` *en la tabla* `t`*. Por ejemplo,*

```
Main> busca P3 parlamento
19
Main> busca P8 parlamento
Program error: no tiene valor en la tabla
```

Solución:

```
busca :: Eq a => a -> Tabla a b -> b
busca x []           = error "no tiene valor en la tabla"
busca x ((x',y):xys)
    | x == x'         = y
    | otherwise       = busca x xys
```

Ejercicio 9.5.7. *Definir la función*

```
busca' :: Eq a => a -> Table a b -> Maybe b
```

tal que `(busca' x t)` *es justo el valor correspondiente a* `x` *en la tabla* `t`*, o* `Nothing` *si* `x` *no tiene valor. Por ejemplo,*

```
busca' P3 parlamento  ⇝  Just 19
busca' P8 parlamento  ⇝  Nothing
```

Solución:

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

Ejercicio 9.5.8. *Comprobar con QuickCheck que si* `(busca' x t)` *es* `Nothing`*, entonces* `x` *es distinto de todos los elementos de* `t`*.*

Solución: La propiedad es

```
prop_BuscaNothing :: Integer -> [(Integer,Integer)] -> Property
prop_BuscaNothing x t =
    busca' x t == Nothing ==>
    x `notElem` [ a | (a,_) <- t ]
```

y la comprobación es

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

Ejercicio 9.5.9. *Comprobar que la función* `busca'` *es equivalente a la función* `lookup` *del Prelude.*

Solución: La propiedad es

```
prop_BuscaEquivLookup :: Integer -> [(Integer,Integer)] -> Bool
prop_BuscaEquivLookup x t =
    busca' x t == lookup x t
```

y la comprobación es

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

Ejercicio 9.5.10. *Definir el tipo* `Coalición` *como una lista de partidos.*

Solución:

```
type Coalición = [Partido]
```

Ejercicio 9.5.11. *Definir la función*

```
mayoría :: Asamblea -> Integer
```

tal que `(mayoría xs)` *es el número de escaños que se necesitan para tener la mayoría en la asamblea* `xs`*. Por ejemplo,*

```
mayoría [(P1,3),(P3,5),(P4,3)]  ⇝  6
```

Solución:

```
mayoría :: Asamblea -> Integer
mayoría xs = ceiling (sum [fromIntegral e | (p,e) <- xs] / 2)
```

Ejercicio 9.5.12. *Definir la función*

```
coaliciones :: Asamblea -> Integer -> [Coalición]
```

tal que `(coaliciones xs n)` *es la lista de coaliciones necesarias para alcanzar* `n` *escaños. Por ejemplo,*

```
coaliciones [(P1,3),(P3,5),(P4,3)] 6  ⇝ [[P3,P4],[P1,P4],[P1,P3]]
coaliciones [(P1,3),(P3,5),(P4,3)] 9  ⇝ [[P1,P3,P4]]
coaliciones [(P1,3),(P3,5),(P4,3)] 14 ⇝ []
coaliciones [(P1,3),(P3,5),(P4,3)] 2  ⇝ [[P4],[P3],[P1]]
```

Solución:

```
coaliciones :: Asamblea -> Integer -> [Coalición]
coaliciones _ n | n <= 0  = [[]]
coaliciones [] n          = []
coaliciones ((p,m):xs) n  =
    coaliciones xs n ++ [p:c | c <- coaliciones xs (n-m)]
```

Ejercicio 9.5.13. *Definir la función*

```
mayorías :: Asamblea -> [Coalición]
```

tal que `(mayorías a)` *es la lista de coaliciones mayoritarias en la asamblea* `a`*. Por ejemplo,*

```
mayorías [(P1,3),(P3,5),(P4,3)]  ⇝  [[P3,P4],[P1,P4],[P1,P3]]
mayorías [(P1,2),(P3,5),(P4,3)]  ⇝  [[P3],[P1,P4],[P1,P3]]
```

Solución:

```
mayorías :: Asamblea -> [Coalición]
mayorías asamblea =
    coaliciones asamblea (mayoría asamblea)
```

Ejercicio 9.5.14. *Definir el tipo de datos* `Asamblea`*.*

Solución:

```
data Asamblea2 = A Asamblea
                 deriving Show
```

Ejercicio 9.5.15. *Definir un generador de datos de tipo* `Asamblea`*. Por ejemplo,*

```
Main> muestra generaAsamblea
A [(P1,1),(P2,1),(P3,0),(P4,1),(P5,0),(P6,1),(P7,0),(P8,1)]
A [(P1,0),(P2,1),(P3,1),(P4,1),(P5,0),(P6,1),(P7,0),(P8,1)]
A [(P1,1),(P2,2),(P3,0),(P4,1),(P5,0),(P6,1),(P7,2),(P8,0)]
A [(P1,1),(P2,0),(P3,1),(P4,0),(P5,0),(P6,1),(P7,1),(P8,1)]
A [(P1,1),(P2,0),(P3,0),(P4,0),(P5,1),(P6,1),(P7,1),(P8,0)]
```

Solución:

```
generaAsamblea :: Gen Asamblea2
generaAsamblea =
    do xs <- listaDe 8 (arbitrary :: Gen Integer)
       return (A (zip [P1,P2,P3,P4,P5,P6,P7,P8] (map abs xs)))

instance Arbitrary Asamblea2 where
    arbitrary   = generaAsamblea
    coarbitrary = undefined
```

Ejercicio 9.5.16. *Definir la propiedad*

```
esMayoritaria :: Coalición -> Asamblea -> Bool
```

tal que `(esMayoritaria c a)` *se verifica si la coalición* `c` *es mayoritaria en la asamblea* `a`*. Por ejemplo,*

```
esMayoritaria [P3,P4] [(P1,3),(P3,5),(P4,3)]  ⇝  True
esMayoritaria [P4] [(P1,3),(P3,5),(P4,3)]     ⇝  False
```

Solución:

```
esMayoritaria :: Coalición -> Asamblea -> Bool
esMayoritaria c a =
    sum [e | (p,e) <- a, p `elem` c] >= (mayoría a)
```

Ejercicio 9.5.17. *Comprobar con QuickCheck que las coaliciones obtenidas por* `(mayorías asamblea)` *son coaliciones mayoritarias en la asamblea.*

Solución: La propiedad es

```
prop_MayoríasSonMayoritarias :: Asamblea2 -> Bool
prop_MayoríasSonMayoritarias (A asamblea) =
  and [esMayoritaria c asamblea | c <- mayorías asamblea]
```

La comprobación es

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

Ejercicio 9.5.18. *Definir la función*

```
esMayoritariaMinimal :: Coalición -> Asamblea -> Bool
```

tal que `(esMayoritariaMinimal c a)` *se verifica si la coalición* `c` *es mayoritaria en la asamblea* `a`*, pero si se quita a* `c` *cualquiera de sus partidos la coalición resultante no es mayoritaria. Por ejemplo,*

```
esMayoritariaMinimal [P3,P4] [(P1,3),(P3,5),(P4,3)]     ↝  True
esMayoritariaMinimal [P1,P3,P4] [(P1,3),(P3,5),(P4,3)]  ↝  False
```

Solución:

```
esMayoritariaMinimal :: Coalición -> Asamblea -> Bool
esMayoritariaMinimal c a =
    esMayoritaria c a &&
    and [not(esMayoritaria (delete p c) a) | p <-c]
```

Ejercicio 9.5.19. *Comprobar con QuickCheck que las coaliciones obtenidas por* `(mayorías asamblea)` *son coaliciones mayoritarias minimales en la asamblea.*

Solución: La propiedad es

```
prop_MayoríasSonMayoritariasMinimales :: Asamblea2 -> Bool
prop_MayoríasSonMayoritariasMinimales (A asamblea) =
  and [esMayoritariaMinimal c asamblea | c <- mayorías asamblea]
```

La comprobación es

```
Main> quickCheck prop_MayoríasSonMayoritariasMinimales
Falsifiable, after 0 tests:
A [(P1,1),(P2,0),(P3,1),(P4,1),(P5,0),(P6,1),(P7,0),(P8,1)]
```

Por tanto, no se cumple la propiedad. Para buscar una coalición no minimal generada por mayorías, definimos la función

```
contraejemplo a =
    head [c | c <- (mayorías a), not(esMayoritariaMinimal c a)]
```

el cálculo del contraejemplo es

```
Main> contraejemplo [(P1,1),(P2,0),(P3,1),(P4,1),(P5,0),(P6,1),(P7,0),(P8,1)]
[P4,P6,P7,P8]
```

La coalición [P4,P6,P7,P8] no es minimal ya que [P4,P6,P8] también es mayoritaria. En efecto,

```
Main> esMayoritaria [P4,P6,P8]
                    [(P1,1),(P2,0),(P3,1),(P4,1),(P5,0),(P6,1),(P7,0),(P8,1)]
True
```

Ejercicio 9.5.20. *Definir la función*

```
coalicionesMinimales :: Asamblea -> Integer -> [Coalición,Escaños]
```

tal que `(coalicionesMinimales xs n)` *es la lista de coaliciones minimales necesarias para alcanzar* `n` *escaños. Por ejemplo,*

```
Main> coalicionesMinimales [(P1,3),(P3,5),(P4,3)] 6
[([P3,P4],8),([P1,P4],6),([P1,P3],8)]
Main> coalicionesMinimales [(P1,3),(P3,5),(P4,3)] 5
[([P3],5),([P1,P4],6)]
```

Solución:

```
coalicionesMinimales :: Asamblea -> Integer -> [(Coalición,Escaños)]
coalicionesMinimales _ n | n <= 0  = [([],0)]
coalicionesMinimales [] n          = []
coalicionesMinimales ((p,m):xs) n  =
    coalicionesMinimales xs n ++
    [(p:ys, t+m) | (ys,t) <- coalicionesMinimales xs (n-m), t<n]
```

Ejercicio 9.5.21. *Definir la función*

```
mayoríasMinimales :: Asamblea -> [Coalición]
```

tal que `(mayoríasMinimales a)` *es la lista de coaliciones mayoritarias minimales en la asamblea* `a`*. Por ejemplo,*

```
Main> mayoríasMinimales [(P1,3),(P3,5),(P4,3)]
[[P3,P4],[P1,P4],[P1,P3]]
```

Solución:

```
mayoríasMinimales :: Asamblea -> [Coalición]
mayoríasMinimales asamblea =
    [c | (c,_) <- coalicionesMinimales asamblea (mayoría asamblea)]
```

Ejercicio 9.5.22. *Comprobar con QuickCheck que las coaliciones obtenidas por*

```
(mayoríasMinimales asamblea)
```

son coaliciones mayoritarias minimales en la asamblea.

Solución: La propiedad es

```
prop_MayoríasMinimalesSonMayoritariasMinimales :: Asamblea2 -> Bool
prop_MayoríasMinimalesSonMayoritariasMinimales (A asamblea) =
  and [esMayoritariaMinimal c asamblea
       | c <- mayoríasMinimales asamblea]
```

La comprobación es

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

9.6. Copia de respaldo

Ejercicio 9.6.1. *Definir la función*

```
copiaFichero :: FilePath -> FilePath -> IO ()
```

tal que `(copiaFichero f1 f2)` *copia, si existe, el fichero* `f1` *en el fichero* `f2`*.*

Solución:

```
copiaFichero :: FilePath -> FilePath -> IO ()
copiaFichero f1 f2 =
    do existe <- doesFileExist f1
       if existe
       then copyFile f1 f2
       else return ()
```

Ejercicio 9.6.2. *Definir la función*

```
respaldo :: IO ()
```

que haga una copia de respaldo del directorio actual en el directorio `/tmp/respaldo`*.*

Solución:

```
respaldo :: IO ()
respaldo =
    do ficheros <- getDirectoryContents "."
       print ficheros
       createDirectory "/tmp/respaldo"
       sequence_ [copiaFichero fichero ("/tmp/respaldo/" ++ fichero)
                  | fichero <- ficheros
                 ]
```

9.7. Ordenación de fichero

Ejercicio 9.7.1. *Definir la función*

```
ordenaFichero :: FilePath -> FilePath -> IO ()
```

tal que `(ordenaFichero f1 f2)` *crea el fichero* `f2` *ordenando las líneas del fichero* `f1`*. Por ejemplo, si el contenido del fichero* `z` *es*

```
García, Juan
Blanco, Ana
Díaz, Pepe
```

entonces `(ordenaFichero "z" "z2")` *crea el fichero* `z2` *con el contenido*

```
Blanco, Ana
Díaz, Pepe
García, Juan
```

Solución:

```
ordenaFichero :: FilePath -> FilePath -> IO ()
ordenaFichero f1 f2 =
    do s <- readFile f1
       writeFile f2 (unlines (sort (lines s)))
```

9.8. Escritura de tablas

Ejercicio 9.8.1. *Ejercicio 12 Definir la función*

```
escribeTabla :: [String] -> IO ()
```

tal que `(escribeTabla xs)` *escribe los elementos de la lista* `xs` *en forma de tabla precedidos por su posición en la lista. Por ejemplo,*

```
Main> escribeTabla ["Juan", "Ana", "Eva"]
1: Juan
2: Ana
3: Eva
```

Solución:

```
escribeTabla :: [String] -> IO ()
escribeTabla xs =
   sequence_ [ putStrLn (show i ++ ": " ++ x)
             | (x,i) <- xs `zip` [1..] ]
```

9.9. Juego interactivo para adivinar un número

En este ejercicio se va a implementar el juego de la adivinación de un número en Haskell. Presentaremos dos versiones dependiendo de si el número lo tiene que adivinar la máquina o el jugador humano.

Ejercicio 9.9.1. *En el primer juego la máquina le pide al jugador humano que piense un número entre 1 y 100 y trata de adivinar el número que ha pensado planteándole conjeturas a las que el jugador humano responde con mayor, menor o exacto según que el número pensado sea mayor, menor o igual que el número conjeturado por la máquina. Definir la función*

```
juego1 :: IO ()
```

para el primer juego. Por ejemplo,

```
Main> juego1
Piensa un numero entre el 1 y el 100.
Es 50? [mayor/menor/exacto] mayor
Es 75? [mayor/menor/exacto] menor
Es 62? [mayor/menor/exacto] mayor
Es 68? [mayor/menor/exacto] exacto
Fin del juego
```

En la definición se usa la función

```
getLine :: IO String
```

para leer la línea de entrada del jugador humano y la función

```
putStrLn :: String -> IO ()
```

para imprimir una línea de texto.
Nota: *Implementar la estrategia de la máquina de forma que se minimice el número de intentos de adivinación.*

Solución:

```
juego1 :: IO ()
juego1 =
    do putStrLn "Piensa un numero entre el 1 y el 100."
       adivina 1 100
       putStrLn "Fin del juego"

adivina :: Int -> Int -> IO ()
adivina a b =
```

```
    do putStr ("Es " ++ show conjetura ++ "? [mayor/menor/exacto] ")
       s <- getLine
       case s of
         "mayor"  -> adivina (conjetura+1) b
         "menor"  -> adivina a (conjetura-1)
         "exacto" -> return ()
         _        -> adivina a b
    where
      conjetura = (a+b) `div` 2
```

Ejercicio 9.9.2. *En el segundo juego la máquina genera un número aleatorio entre 1 y 100 y le pide al jugador humano que adivine el número que ha pensado planteándole conjeturas a las que la máquina responde con mayor, menor o exacto según que el número pensado sea mayor, menor o igual que el número conjeturado por el jugador humano. Definir la función*

```
juego2 :: IO ()
```

para el segundo juego. Por ejemplo,

```
Main> juego2
Tienes que adivinar un numero entre 1 y 100
Escribe un numero: 50
 es bajo.
Escribe un numero: 75
 es alto.
Escribe un numero: 62
 es alto.
Escribe un numero: 56
 es bajo.
Escribe un numero: 59
 es bajo.
Escribe un numero: 60
 Exactamente
```

Solución:

```
juego2 :: IO ()
juego2 = do
  hSetBuffering stdout NoBuffering
  n <- randomRIO (1::Int, 100)
  putStrLn "Tienes que adivinar un numero entre 1 y 100"
  adivina' n
```

```
adivina' :: Int -> IO ()
adivina' n =
    do putStr "Escribe un numero: "
       c <- getLine
       let x = read c
       case (compare x n) of
         LT -> do putStrLn " es bajo."
                  adivina' n
         GT -> do putStrLn " es alto."
                  adivina' n
         EQ -> do putStrLn " Exactamente"
```

Capítulo 10

Tipos de datos recursivos

Contenido

Los ejercicios de este capítulo se corresponden con las clases de la cuarta semana.

10.1. Directorios

Ejercicio 10.1.1. *Un fichero contiene datos o es un directorio. Un directorio contiene otros ficheros (que a su vez pueden ser directorios).*
Definir el tipo de datos `Fichero` *para representar los ficheros. Por ejemplo,*

```
[ Fichero "apa",
  Dir "bepa" [Fichero "apa", Dir "bepa" [], Dir "cepa" [Fichero "bepa" ]],
  Dir "cepa" [Dir "bepa" [], Dir "cepa" [Fichero "apa"]]]
```

representa un sistema de fichero compuesto por el fichero `apa` *y los directorios* `bepa` *y* `cepa`*. El directorio* `bepa` *contiene el fichero* `bepa/apa` *y los directorios* `bepa/bepa` *y* `bepa/cepa`*. El directorio* `bepa/bepa` *está vacío y el* `bepa/cepa` *contiene el fichero* `bepa/cepa/bepa`*.*

Solución:

```
data Fichero = Fichero String
             | Dir String [Fichero]
             deriving (Eq, Show)

type SistemaFicheros = [Fichero]
```

Ejercicio 10.1.2. *Definir* `ejemploSistemaFicheros` *para representar el ejemplo de sistema de ficheros del ejercicio anterior.*

Solución:

```
ejemploSistemaFicheros :: SistemaFicheros
ejemploSistemaFicheros =
  [Fichero "apa",
   Dir "bepa" [Fichero "apa", Dir "bepa" [], Dir "cepa" [Fichero "bepa"]],
   Dir "cepa" [Dir "bepa" [], Dir "cepa" [Fichero "apa"]]]
```

Ejercicio 10.1.3. *Definir la función*

```
busca :: SistemaFicheros -> String -> [String]
```

tal que `(busca s f)` *es la lista de las ocurrencias del fichero* `f` *en el sistema de ficheros* `s`*. Por ejemplo,*

```
Main> busca ejemploSistemaFicheros "apa"
["apa","bepa/apa","cepa/cepa/apa"]
Main> busca ejemploSistemaFicheros "bepa"
["bepa/cepa/bepa"]
Main> busca ejemploSistemaFicheros "cepa"
[]
```

Solución:

```
busca :: SistemaFicheros -> String -> [String]
busca s f =
    [f | Fichero f' <- s, f == f'] ++
    [dir ++ "/" ++ camino | Dir dir s' <- s, camino <- busca s' f]
```

10.2. Lógica proposicional

Ejercicio 10.2.1. *Una fórmula proposicional es una expresión de una de las siguientes formas:*

- *una variable proposicional (una cadena)*
- `p & q` *(conjunción)*
- `p | q` *(disyunción)*
- `~p` *(negación)*

donde `p` *y* `q` *son fórmulas proposicionales. Por ejemplo,* `p | ~p` *es una fórmula proposicional. Definir el tipo de datos* `Prop` *para representar las fórmulas proposicionales. Representar las conectivas por* `:&`, `:|` *y* `No`*. Por tanto, el ejemplo anterior se representa por*

```
Var "p" :| No (Var "p")
```

Solución:

```
data Prop  = Var Nombre
           | Prop :& Prop
           | Prop :| Prop
           | No Prop
           deriving ( Eq, Show )

type Nombre = String
```

Ejercicio 10.2.2. *Definir la función*

```
vars :: Prop -> [Nombre]
```

tal que `(vars f)` *es el conjunto de las variables proposicionales de la fórmula* `f`*. Por ejemplo,*

```
vars (Var "p" :& (Var "q" :| Var "p"))  ⇝  ["p","q"]
```

Solución:

```
vars :: Prop -> [Nombre]
vars (Var x)  = [x]
vars (a :& b) = vars a `union` vars b
vars (a :| b) = vars a `union` vars b
vars (No a)   = vars a
```

Ejercicio 10.2.3. *Una interpretación es una lista de pares formados por el nombre de una variable y un valor booleano. Por ejemplo,*

```
[("p",True),("q",False)]
```

es una interpretación. Definir la función

```
valor :: Prop -> [(String,Bool)] -> Bool
```

tal que `(valor f i)` *es el valor de la fórmula* `f` *en la interpretación* `i`*. Por ejemplo,*

```
Main> valor (Var "p" :& (Var "q" :| Var "p")) [("p",True),("q",False)]
True
Main> valor (Var "p" :& (Var "q" :| Var "p")) [("p",False),("q",False)]
False
```

Solución:

```
valor :: Prop -> [(Nombre,Bool)] -> Bool
valor (Var x)  i = fromJust (lookup x i)
valor (a :& b) i = valor a i && valor b i
valor (a :| b) i = valor a i || valor b i
valor (No a)   i = not (valor a i)
```

Ejercicio 10.2.4. *Definir la función*

```
interpretaciones :: [Nombre] -> [[(Nombre,Bool)]]
```

tal que `(interpretaciones xs)` *es la lista de todas las interpretaciones correspondiente a la lista de nombres* `xs`*. Por ejemplo,*

```
Main> interpretaciones ["x","y"]
[[("x",False),("y",False)],
 [("x",False),("y",True)],
 [("x",True),("y",False)],
 [("x",True),("y",True)]]
```

Solución:

```
interpretaciones :: [Nombre] -> [[(Nombre,Bool)]]
interpretaciones []     = [[]]
interpretaciones (x:xs) = [ (x,v):i | v <- [False,True],
                                      i <- interpretaciones xs]
```

Ejercicio 10.2.5. *Definir la función*

```
tautología :: Prop -> Bool
```

tal que `(tautología f)` *se verifica si la fórmula* `f` *es una tautología; es decir, el valor de* `f` *es verdadero en todas las interpretaciones. Por ejemplo,*

```
Main> tautología (Var "p" :| (No (Var "p")))
True
Main> tautología (Var "p" :& (Var "q" :| Var "p"))
False
```

Solución:

```
tautología :: Prop -> Bool
tautología f =
    and [valor f i | i <- interpretaciones (vars f)]
```

10.3. Expresiones aritméticas

Ejercicio 10.3.1. *Definir el tipo de datos* `Expr` *para representar las expresiones aritmética formadas por números enteros, sumas y productos. Por ejemplo, la expresión*

```
Sum (Núm 3) (Pro (Núm 4) (Núm 6)) :: Expr
```

*representa 3+(4*6).*

Solución:

```
data Expr = Núm Integer
          | Sum Expr Expr
          | Pro Expr Expr
          deriving Eq
```

Ejercicio 10.3.2. *Definir la función*

```
muestraExpr :: Expr -> String
```

tal que `(muestraExpr e)` *es la cadena correspondiente a la expresión* `e`*. Por ejemplo,*

```
muestraExpr (Sum (Núm 3) (Pro (Núm 4) (Núm 6)))  ⇝  "3+4*6"
muestraExpr (Pro (Núm 3) (Sum (Núm 4) (Núm 6)))  ⇝  "3*(4+6)"
```

Hacer `Expr` *subclase de* `Show` *usando* `muestraExpr` *como* `show`*.*

Solución:

```
muestraExpr :: Expr -> String
muestraExpr (Núm n)
    | n<0             = "(" ++ show n ++ ")"
    | otherwise       = show n
```

```
muestraExpr (Sum a b) = muestraExpr a ++ "+" ++ muestraExpr b
muestraExpr (Pro a b) = muestraFactor a ++ "*" ++ muestraFactor b

muestraFactor :: Expr -> String
muestraFactor (Sum a b) = "(" ++ muestraExpr (Sum a b) ++ ")"
muestraFactor e         = muestraExpr e

instance Show Expr where
    show = muestraExpr
```

Ejercicio 10.3.3. *Definir la función*

```
valor :: Expr -> Integer
```

tal que `(valor e)` *es el valor de la expresión* `e`*. Por ejemplo,*

```
valor (Sum (Núm 3) (Pro (Núm 4) (Núm 6)))  ⇝  27
```

Solución:

```
valor :: Expr -> Integer
valor (Núm n)   = n
valor (Sum a b) = valor a + valor b
valor (Pro a b) = valor a * valor b
```

Ejercicio 10.3.4. *Definir el generador*

```
arbExpr :: Int -> Gen Expr
```

que genere expresiones arbitrarias. Por ejemplo,

```
Main> muestra (arbExpr 10)
(-1)*1*0+1*1+(-1)+1
(2+(-2))*(1+(-2))*(0+(-1)*(-2))+(-1)*0*(-2)*(-1)*(-2)
0
1*((-2)+3*2)*(3*3*3+1+1*3)
2*(((-1)+0)*((-2)+0)+0*2*(1+2))
```

Usar `arbExpr` *para hacer* `Expr` *subclase de* `Arbitrary`*.*

Solución:

```
arbExpr :: Int -> Gen Expr
arbExpr s =
    frequency [ (1, do n <- arbitrary
                        return (Núm n))
              , (s, do a <- arbExpr s'
                        b <- arbExpr s'
                        return (Sum a b))
              , (s, do a <- arbExpr s'
                        b <- arbExpr s'
                        return (Pro a b))
              ]
    where s' = s `div` 2

instance Arbitrary Expr where
    arbitrary   = sized arbExpr
    coarbitrary = undefined
```

Ejercicio 10.3.5. *Definir la función*

```
preguntas :: IO ( )
```

de forma que genere expresiones aritméticas, pregunte por su valor y compruebe las respuestas. Por ejemplo.

```
Main> preguntas
Calcula el valor de (-3)+1+3+(-2) : -1
Es correcto
Calcula el valor de ((-1)+0)*(-4)*3+(-4)*(0+3) : 3
Es incorrecto
Calcula el valor de 0 : Interrupted!
```

Solución:

```
preguntas :: IO ( )
preguntas =
    do rnd <- newStdGen
       let e = generate 5 rnd arbitrary
       putStr ("Calcula el valor de " ++ show e ++ " : ")
       respuesta <- getLine
       putStrLn (if read respuesta == valor e
                 then "Es correcto"
                 else "Es incorrecto")
       preguntas
```

Ejercicio 10.3.6. *Definir la función*

```
operadores :: Expr -> Int
```

tal que `(operadores e)` *es el número de operadores en la expresión* `e`*. Por ejemplo,*

```
Main> operadores (Sum (Núm 2) (Pro (Núm 3) (Núm 4)))
2
```

Solución:

```
operadores :: Expr -> Int
operadores (Núm _)   = 0
operadores (Sum a b) = 1 + operadores a + operadores b
operadores (Pro a b) = 1 + operadores a + operadores b
```

10.4. Expresiones aritméticas generales

En este ejercicio se generaliza el anterior añadiendo las operaciones de restar y dividir.

Ejercicio 10.4.1. *Definir el tipo de datos* `Expr` *para representar las expresiones aritmética formadas por números enteros, sumas, restas, productos y divisiones. Por ejemplo, la expresión*

```
Sum (Núm 3) (Pro (Núm 4) (Núm 6)) :: Expr
```

*representa 3+(4*6).*

Solución:

```
data Expr = Núm Integer
          | Sum Expr Expr
          | Res Expr Expr
          | Pro Expr Expr
          | Div Expr Expr
          deriving Eq
```

Ejercicio 10.4.2. *Definir la función*

```
muestraExpr :: Expr -> String
```

tal que `(muestraExpr e)` *es la cadena correspondiente a la expresión* `e`*. Por ejemplo,*

```
muestraExpr (Res (Núm 3) (Div (Núm 4) (Núm 6)))  ⇝  "3-4/6"
```

Hacer `Expr` *subclase de* `Show` *usando* `muestraExpr` *como* `show`*.*

Solución:

```
muestraExpr :: Expr -> String
muestraExpr (Núm n)
    | n<0             = "(" ++ show n ++ ")"
    | otherwise       = show n
muestraExpr (Sum a b) = muestraExpr a ++ "+" ++ muestraExpr b
muestraExpr (Res a b) = muestraExpr a ++ "-" ++ muestraExpr b
muestraExpr (Pro a b) = muestraFactor a ++ "*" ++ muestraFactor b
muestraExpr (Div a b) = muestraFactor a ++ "*" ++ muestraFactor b

muestraFactor :: Expr -> String
muestraFactor (Sum a b) = "(" ++ muestraExpr (Sum a b) ++ ")"
muestraFactor (Res a b) = "(" ++ muestraExpr (Res a b) ++ ")"
muestraFactor e         = muestraExpr e

instance Show Expr where
    show = muestraExpr
```

Ejercicio 10.4.3. *Definir la función*

```
valor :: Expr -> Integer
```

tal que `(valor e)` *es el valor de la expresión* `e`*. Por ejemplo,*

```
valor (Sum (Núm 3) (Pro (Núm 4) (Núm 6)))  ⇝  27
```

Solución:

```
valor :: Expr -> Integer
valor (Núm n)   = n
valor (Sum a b) = valor a + valor b
valor (Res a b) = valor a - valor b
valor (Pro a b) = valor a * valor b
valor (Div a b) = valor a `div` valor b
```

Ejercicio 10.4.4. *Definir el generador*

```
arbExpr :: Int -> Gen Expr
```

que genere expresiones arbitrarias. Por ejemplo,

```
Main> muestra (arbExpr 10)
(-1)*1*0+1*1+(-1)+1
(2+(-2))*(1+(-2))*(0+(-1)*(-2))+(-1)*0*(-2)*(-1)*(-2)
0
1*((-2)+3*2)*(3*3*3+1+1*3)
2*(((-1)+0)*((-2)+0)+0*2*(1+2))
```

Usar `arbExpr` *para hacer* `Expr` *subclase de* `Arbitrary`.

Solución:

```
arbExpr :: Int -> Gen Expr
arbExpr s =
    frequency [ (1, do n <- arbitrary
                        return (Núm n))
              , (s, do a <- arbExpr s'
                        b <- arbExpr s'
                        return (Sum a b))
              , (s, do a <- arbExpr s'
                        b <- arbExpr s'
                        return (Res a b))
              , (s, do a <- arbExpr s'
                        b <- arbExpr s'
                        return (Pro a b))
              , (s, do a <- arbExpr s'
                        b <- arbExpr s'
                        return (Div a b))
              ]
    where s' = s `div` 2

instance Arbitrary Expr where
    arbitrary   = sized arbExpr
    coarbitrary = undefined
```

Ejercicio 10.4.5. *Definir la función*

```
preguntas :: IO ( )
```

de forma que genere expresiones aritméticas, pregunte por su valor y compruebe las respuestas. Por ejemplo.

```
Main> preguntas
Calcula el valor de (-3)+1+3+(-2) : -1
```

```
Es correcto
Calcula el valor de ((-1)+0)*(-4)*3+(-4)*(0+3) : 3
Es incorrecto
Calcula el valor de 0 : Interrupted!
```

Solución:

```
preguntas :: IO ( )
preguntas =
    do rnd <- newStdGen
       let e = generate 5 rnd arbitrary
       putStr ("Calcula el valor de " ++ show e ++ " : ")
       respuesta <- getLine
       putStrLn (if read respuesta == valor e
                 then "Es correcto"
                 else "Es incorrecto")
       preguntas
```

Ejercicio 10.4.6. *Definir la función*

```
operadores :: Expr -> Int
```

tal que `(operadores e)` *es el número de operadores en la expresión* `e`*. Por ejemplo,*

```
Main> operadores (Sum (Núm 2) (Pro (Núm 3) (Núm 4)))
2
```

Solución:

```
operadores :: Expr -> Int
operadores (Núm _)   = 0
operadores (Sum a b) = 1 + operadores a + operadores b
operadores (Res a b) = 1 + operadores a + operadores b
operadores (Pro a b) = 1 + operadores a + operadores b
operadores (Div a b) = 1 + operadores a + operadores b
```

10.5. Expresiones aritméticas generales con operadores

En este ejercicio se hace una representación alternativa de las expresiones aritméticas definiendo los operadores.

Ejercicio 10.5.1. *Definir el tipo de datos* `Ops` *para representar los operadores aritméticos (suma, resta, producto y división) y* `Expr` *para representar las expresiones aritmética formadas por números enteros, sumas, restas, productos y divisiones. Por ejemplo, la expresión*

```
Op Sum (Núm 3) (Op Pro (Núm 4) (Núm 6)) :: Expr
```

*representa 3+(4*6).*

Solución:

```
data Expr = Núm Integer
          | Op Ops Expr Expr
          deriving (Eq)

data Ops = Sum | Res | Pro | Div
           deriving ( Eq, Show )
```

Ejercicio 10.5.2. *Definir la función*

```
muestraExpr :: Expr -> String
```

tal que `(muestraExpr e)` *es la cadena correspondiente a la expresión* `e`*. Por ejemplo,*

```
muestraExpr (Op Sum (Núm 3) (Op Pro (Núm 4) (Núm 6)))  ⇝  "3-4/6"
```

Hacer `Expr` *subclase de* `Show` *usando* `muestraExpr` *como* `show`*.*

Solución:

```
muestraExpr :: Expr -> String
muestraExpr (Núm n)
    | n<0             = "(" ++ show n ++ ")"
    | otherwise       = show n
muestraExpr (Op Sum a b) = muestraExpr a ++ "+" ++ muestraExpr b
muestraExpr (Op Res a b) = muestraExpr a ++ "-" ++ muestraExpr b
muestraExpr (Op Pro a b) = muestraFactor a ++ "*" ++ muestraFactor b
muestraExpr (Op Div a b) = muestraFactor a ++ "*" ++ muestraFactor b

muestraFactor :: Expr -> String
muestraFactor (Op Sum a b) = "(" ++ muestraExpr (Op Sum a b) ++ ")"
muestraFactor (Op Res a b) = "(" ++ muestraExpr (Op Res a b) ++ ")"
muestraFactor e            = muestraExpr e

instance Show Expr where
    show = muestraExpr
```

Ejercicio 10.5.3. *Definir la función*

```
valor :: Expr -> Integer
```

tal que `(valor e)` *es el valor de la expresión* `e`*. Por ejemplo,*

```
valor (Op Sum (Núm 3) (Op Pro (Núm 4) (Núm 6)))  ⇝  27
```

Solución:

```
valor :: Expr -> Integer
valor (Núm n)   = n
valor (Op op a b) = valorOp op (valor a) (valor b)
    where
      valorOp Sum a b = a + b
      valorOp Res a b = a - b
      valorOp Pro a b = a * b
      valorOp Div a b = a `div` b
```

Ejercicio 10.5.4. *Definir el generador*

```
arbExpr :: Int -> Gen Expr
```

que genere expresiones arbitrarias. Por ejemplo,

```
Main> muestra (arbExpr 10)
0*(0-0)*(0+0)*0*0*(0-0*0*(0+0))
(-5)*(-4)-2-(-2)-(-5)+0*(-2)-4*5+(-4)+(-3)-2*0
3*0+(-2)-(-1)-3-(-1)-2*(-3)-(1+(-1))*(-3)*(-1)-2*0-(-3)*0
((-4)*(-1)-(-4)*(-1)-(-3)*1-0*3)*((-5)*(-2)*((-5)+(-1))+(-4)*0*((-3)-1))
((-1)*0+(-1)+0-0-(-1)-1)*(-1)*(-1)*(1-(-1))*1*(-1)*(-1)
```

Usar `arbExpr` *para hacer* `Expr` *subclase de* `Arbitrary`*.*

Solución:

```
arbExpr :: Int -> Gen Expr
arbExpr s =
    frequency [ (1, do n <- arbitrary
                        return (Núm n))
              , (s, do a <- arbExpr s'
                        b <- arbExpr s'
                        return (Op Sum a b))
              , (s, do a <- arbExpr s'
                        b <- arbExpr s'
                        return (Op Res a b))
              , (s, do a <- arbExpr s'
                        b <- arbExpr s'
```

```
                    return (Op Pro a b))
             , (s, do a <- arbExpr s'
                    b <- arbExpr s'
                    return (Op Div a b))
             ]
    where s' = s `div` 2

instance Arbitrary Expr where
    arbitrary   = sized arbExpr
    coarbitrary = undefined
```

Ejercicio 10.5.5. *Definir la función*

```
preguntas :: IO ( )
```

de forma que genere expresiones aritméticas, pregunte por su valor y compruebe las respuestas. Por ejemplo.

```
Main> preguntas
Calcula el valor de (-4) : -4
Es correcto
Calcula el valor de 3-1*(-3) : 6
Es correcto
```

Solución:

```
preguntas :: IO ( )
preguntas =
    do rnd <- newStdGen
       let e = generate 5 rnd arbitrary
       putStr ("Calcula el valor de " ++ show e ++ " : ")
       respuesta <- getLine
       putStrLn (if read respuesta == valor e
                 then "Es correcto"
                 else "Es incorrecto")
       preguntas
```

Ejercicio 10.5.6. *Definir la función*

```
operadores :: Expr -> Int
```

tal que `(operadores e)` *es el número de operadores en la expresión* `e`*. Por ejemplo,*

```
Main> operadores (Op Sum (Núm 3) (Op Pro (Núm 4) (Núm 6)))
2
```

Solución:

```
operadores :: Expr -> Int
operadores (Núm _)   = 0
operadores (Op op a b) = 1 + operadores a + operadores b
```

10.6. Expresiones aritméticas con notación infija

En este ejercicio se hace una nueva presentación de la expresiones aritméticas usando notación infija.

Ejercicio 10.6.1. *Definir el tipo de datos* `Expr` *para representar las expresiones aritmética formadas por números enteros, sumas y productos. Por ejemplo, la expresión*

```
N 3 :+: (N 4 :*: N 6) :: Expr
```

*representa 3+(4*6).*

Solución:

```
data Expr = N Integer
          | Expr :+: Expr
          | Expr :*: Expr
          deriving Eq
```

Ejercicio 10.6.2. *Definir la función*

```
muestraExpr :: Expr -> String
```

tal que `(muestraExpr e)` *es la cadena correspondiente a la expresión* `e`*. Por ejemplo,*

```
muestraExpr (N 3 :+: (N 4 :*: N 6))  ⇝  "3+4*6"
muestraExpr (N 3 :*: (N 4 :+: N 6))  ⇝  "3*(4+6)"
```

Hacer `Expr` *subclase de* `Show` *usando* `muestraExpr` *como* `show`*.*

Solución:

```
muestraExpr :: Expr -> String
muestraExpr (N n)
    | n<0             = "(" ++ show n ++ ")"
    | otherwise       = show n
```

```
muestraExpr (a :+: b) = muestraExpr a ++ "+" ++ muestraExpr b
muestraExpr (a :*: b) = muestraFactor a ++ "*" ++ muestraFactor b

muestraFactor :: Expr -> String
muestraFactor (a :+: b) = "(" ++ muestraExpr (a :+: b) ++ ")"
muestraFactor e         = muestraExpr e

instance Show Expr where
    show = muestraExpr
```

Ejercicio 10.6.3. *Definir la función*

```
valor :: Expr -> Integer
```

tal que `(valor e)` *es el valor de la expresión* `e`*. Por ejemplo,*

```
valor (N 3 :+: (N 4 :*: N 6))  ⇝  27
```

Solución:

```
valor :: Expr -> Integer
valor (N n)   = n
valor (a :+: b) = valor a + valor b
valor (a :*: b) = valor a * valor b
```

Ejercicio 10.6.4. *Definir el generador*

```
arbExpr :: Int -> Gen Expr
```

que genere expresiones arbitrarias. Por ejemplo,

```
Main> muestra (arbExpr 10)
(-1)*1*0+1*1+(-1)+1
(2+(-2))*(1+(-2))*(0+(-1)*(-2))+(-1)*0*(-2)*(-1)*(-2)
0
1*((-2)+3*2)*(3*3*3+1+1*3)
2*(((-1)+0)*((-2)+0)+0*2*(1+2))
```

Usar `arbExpr` *para hacer* `Expr` *subclase de* `Arbitrary`*.*

Solución:

```
arbExpr :: Int -> Gen Expr
arbExpr s =
    frequency [ (1, do n <- arbitrary
                        return (N n))
              , (s, do a <- arbExpr s'
                        b <- arbExpr s'
                        return (a :+: b))
              , (s, do a <- arbExpr s'
                        b <- arbExpr s'
                        return (a :*: b))
              ]
    where s' = s `div` 2

instance Arbitrary Expr where
    arbitrary   = sized arbExpr
    coarbitrary = undefined
```

Ejercicio 10.6.5. *Definir la función*

```
preguntas :: IO ( )
```

de forma que genere expresiones aritméticas, pregunte por su valor y compruebe las respuestas. Por ejemplo.

```
Main> preguntas
Calcula el valor de (-3)+1+3+(-2) : -1
Es correcto
Calcula el valor de ((-1)+0)*(-4)*3+(-4)*(0+3) : 3
Es incorrecto
Calcula el valor de 0 : Interrupted!
```

Solución:

```
preguntas :: IO ( )
preguntas =
    do rnd <- newStdGen
       let e = generate 5 rnd arbitrary
       putStr ("Calcula el valor de " ++ show e ++ " : ")
       respuesta <- getLine
       putStrLn (if read respuesta == valor e
                 then "Es correcto"
                 else "Es incorrecto")
       preguntas
```

Ejercicio 10.6.6. *Definir la función*

```
operadores :: Expr -> Int
```

tal que `(operadores e)` *es el número de operadores en la expresión* `e`*. Por ejemplo,*

```
Main> operadores (N 2 :+: (N 3 :*: N 4))
2
```

Solución:

```
operadores :: Expr -> Int
operadores (N _)   = 0
operadores (a :+: b) = 1 + operadores a + operadores b
operadores (a :*: b) = 1 + operadores a + operadores b
```

10.7. Expresiones aritméticas con variables y derivación simbólica

Ejercicio 10.7.1. *Definir el tipo de datos* `Expr` *para representar las expresiones aritmética formadas por números enteros, sumas, productos y variable. Por ejemplo, la expresión*

```
Sum (Núm 3) (Pro (Núm 4) (Var "a")) :: Expr
```

*representa 3+(4*a).*

Solución:

```
data Expr = Núm Integer
          | Sum Expr Expr
          | Pro Expr Expr
          | Var Nombre
          deriving ( Eq )

type Nombre = String
```

Ejercicio 10.7.2. *Definir la función*

```
muestraExpr :: Expr -> String
```

tal que `(muestraExpr e)` *es la cadena correspondiente a la expresión* `e`*. Por ejemplo,*

```
muestraExpr (Sum (Núm 3) (Pro (Núm 4) (Var "a")))  ⇝  "3+4*a"
muestraExpr (Pro (Núm 3) (Sum (Núm 4) (Var "a")))  ⇝  "3*(4+a)"
```

Hacer `Expr` *subclase de* `Show` *usando* `muestraExpr` *como* `show`.

Solución:

```
muestraExpr :: Expr -> String
muestraExpr (Núm n)
    | n<0             = "(" ++ show n ++ ")"
    | otherwise       = show n
muestraExpr (Sum a b) = muestraExpr a ++ "+" ++ muestraExpr b
muestraExpr (Pro a b) = muestraFactor a ++ "*" ++ muestraFactor b
muestraExpr (Var x)   = x

muestraFactor :: Expr -> String
muestraFactor (Sum a b) = "(" ++ muestraExpr (Sum a b) ++ ")"
muestraFactor e         = muestraExpr e

instance Show Expr where
    show = muestraExpr
```

Ejercicio 10.7.3. *Un entorno es una lista de pares formados por una cadena y un número entero. Definir el tipo* `Ent` *para representar los entornos.*

Solución:

```
type Ent = [(String,Integer)]
```

Ejercicio 10.7.4. *Definir la función*

```
valor :: Ent -> Expr -> Integer
```

tal que `(valor e x)` *es el valor de la expresión* `x` *en el entorno* `e`. *Por ejemplo,*

```
valor [("a",6)] (Sum (Núm 3) (Pro (Núm 4) (Var "a")))  ⇝  27
```

Solución:

```
valor :: Ent -> Expr -> Integer
valor ent (Núm n)   = n
valor ent (Sum a b) = valor ent a + valor ent b
valor ent (Pro a b) = valor ent a * valor ent b
valor ent (Var x)   = fromJust (lookup x ent)
```

Ejercicio 10.7.5. *Definir la función*

```
vars :: Expr -> [Nombre]
```

tal que `(vars e)` *es la lista de las variables en la expresión* `e`*. Por ejemplo,*

```
vars (Sum (Var "x") (Pro (Núm 4) (Var "a")))    ⇝  ["x","a"]
vars (Sum (Var "x") (Pro (Var "a") (Var "a")))  ⇝  ["x","a"]
```

Solución:

```
vars :: Expr -> [Nombre]
vars (Núm n)   = []
vars (Sum a b) = vars a `union` vars b
vars (Pro a b) = vars a `union` vars b
vars (Var y)   = [y]
```

Ejercicio 10.7.6. *Definir la función*

```
derivada :: Expr -> Nombre -> Expr
```

tal que `(derivada e x)` *es la derivada de la expresión* `e` *respecto de la variable* `x`*. Por ejemplo,*

```
Main> derivada (Pro (Núm 2) (Var "x")) "x"
2*1+x*0
```

Solución:

```
derivada :: Expr -> Nombre -> Expr
derivada (Núm n)   x = Núm 0
derivada (Sum a b) x = Sum (derivada a x) (derivada b x)
derivada (Pro a b) x = Sum (Pro a (derivada b x)) (Pro b (derivada a x))
derivada (Var y)   x
    | x == y       = Núm 1
    | otherwise    = Núm 0
```

Ejercicio 10.7.7. *Definir el generador*

```
arbExpr :: Int -> Gen Expr
```

que genere expresiones arbitrarias. Por ejemplo,

```
Main> muestra (arbExpr 5)
0*0*(0+0)*0*0
(z+(-1))*x*y*y*0
y*(z+x)
(y+y)*y*(-5)+0
0
```

Usar `arbExpr` *para hacer* `Expr` *subclase de* `Arbitrary`.

Solución:

```
arbExpr :: Int -> Gen Expr
arbExpr s =
    frequency [ (1, do n <- arbitrary
                        return (Núm n))
              , (s, do a <- arbExpr s'
                        b <- arbExpr s'
                        return (Sum a b))
              , (s, do a <- arbExpr s'
                        b <- arbExpr s'
                        return (Pro a b))
              , (1, do x <- oneof [ return s | s <- ["x","y","z"] ]
                        return (Var x))
              ]
    where s' = s `div` 2

instance Arbitrary Expr where
    arbitrary   = sized arbExpr
    coarbitrary = undefined
```

Ejercicio 10.7.8. *Comprobar con QuickCheck que todas la variables de la derivada de una expresión son variables de la expresión.*

Solución: La propiedad es

```
prop_DerivadaVars e =
  and [ y `elem` xs | y <- ys ]
 where
  xs = vars e
  ys = vars (derivada e "x")
```

La comprobación es

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

Ejercicio 10.7.9. *Definir el generador*

```
arbExpDer :: Gen (Expr,Expr)
```

que genere expresiones arbitrarias junto con sus derivadas respecto de `x`. *Por ejemplo,*

```
Main> muestra arbExpDer
(y,0)
(x*(1*z+1+z),x*(1*0+z*0+0+0)+(1*z+1+z)*1)
(0*(x+(-1)),0*(1+0)+(x+(-1))*0)
(y*z*z*5+(-1)*y*5,y*z*(z*0+5*0)+z*5*(y*0+z*0)+(-1)*(y*0+5*0)+y*5*0)
((-2)+y+(-1)+3,0+0+0+0)
```

Solución:

```
arbExpDer :: Gen (Expr,Expr)
arbExpDer =
  do a <- arbitrary
     return (a, derivada a "x")
```

Ejercicio 10.7.10. *Los polinomios pueden representarse mediante listas de pares formados por un número entero y una lista de variables. Por ejemplo, el polinomio*

```
x + 3*x*y + 2*x*x + 5*y
```

puede representarse mediante la lista

```
[(1,["x"]), (3,["x","y"]), (2,["x","x"]), (5,["y"])]
```

Definir el tipo `Poli` *para representar a los polinomios.*

Solución:

```
type Poli = [(Integer,[Nombre])]
```

Ejercicio 10.7.11. *Definir la función*

```
poli :: Expr -> Poli
```

tal que `(poli e)` *es el polinomio correspondiente a la expresión* `e`*. Por ejemplo,*

```
Main> poli (Núm 0)                 ⇝  []
Main> poli (Núm 3)                 ⇝  [(3,[])]
Main> poli (Var "y")               ⇝  [(1,["y"])]
Main> poli (Sum (Núm 3) (Var "y")) ⇝  [(3,[]),(1,["y"])]
Main> poli (Sum (Núm 3) (Núm 2))   ⇝  [(3,[]),(2,[])]
Main> poli (Pro (Núm 3) (Var "y")) ⇝  [(3,["y"])]
Main> poli (Pro (Núm 3) (Núm 2))   ⇝  [(6,[])]
```

Solución:

```
poli :: Expr -> Poli
poli (Núm 0)   = []
poli (Núm n)   = [(n,[])]
poli (Var x)   = [(1,[x])]
poli (Sum a b) = poli a ++ poli b
poli (Pro a b) = [ (ca*cb,xa++xb) | (ca,xa) <- poli a, (cb,xb) <- poli b ]
```

Ejercicio 10.7.12. *Definir la función*

```
simpPoli :: Poli -> Poli
```

tal que `(simpPoli p)` *es el polinomio obtenido simplificando el polinomio* `p`*. Por ejemplo,*

```
Main> simpPoli [(1,["x","y"]),(3,["x"]),(5,["y","x"])]
[(3,["x"]),(6,["x","y"])]
```

Solución:

```
simpPoli :: Poli -> Poli
simpPoli p = [(sum [n | (n,x') <- p', x == x'], x) | x <- xs]
    where
      p' = [(n,sort x) | (n,x) <- p]
      xs = sort (nub [x | (_,x) <- p'])
```

Ejercicio 10.7.13. *Definir la función*

```
expr :: Poli -> Expr
```

tal que `(expr p)` *es la expresión correspondiente al polinomio* `p`*. Por ejemplo,*

```
Main> expr [(1,["x"]),(3,["y"])]
x+3*y
```

Solución:

```
expr :: Poli -> Expr
expr p =
    sum [pro n x | (n,x) <- p, n /= 0]
    where
      pro n []  = Núm n
      pro 1 xs  = pros xs
      pro n xs  = Pro (Núm n) (pros xs)

      pros []     = Núm 1
```

```
pros [a]    = Var a
pros (a:as) = Pro (Var a) (pros as)

sum []     = Núm 0
sum [a]    = a
sum (a:as) = Sum a (sum as)
```

Ejercicio 10.7.14. *Definir la función*

```
simplifica :: Expr -> Expr
```

tal que `(simplifica e)` *es la expresión obtenidad simplificando la expresión* `e`*. Por ejemplo,*

```
Main> Sum (Núm 2) (Núm 3)
2+3
Main> simplifica (Sum (Núm 2) (Núm 3))
5
Main> Sum (Sum (Pro (Núm 2) (Var "x")) (Núm 4))
          (Sum (Pro (Núm 3) (Var "x")) (Núm 5))
2*x+4+3*x+5
Main> simplifica (Sum (Sum (Pro (Núm 2) (Var "x")) (Núm 4))
                      (Sum (Pro (Núm 3) (Var "x")) (Núm 5)))
9+5*x
Main> Sum (Sum (Pro (Núm 0) (Var "x")) (Núm (-2)))
          (Sum (Pro (Núm 3) (Var "y")) (Núm 5))
0*x+(-2)+3*y+5
Main> simplifica (Sum (Sum (Pro (Núm 0) (Var "x")) (Núm (-2)))
                  (Sum (Pro (Núm 3) (Var "y")) (Núm 5)))
3+3*y
```

Solución:

```
simplifica :: Expr -> Expr
simplifica e = expr (simpPoli (poli e))
```

Ejercicio 10.7.15. *Definir la función*

```
esNúm :: Expr -> Bool
```

tal que `(esNum e)` *se verifica si la expresión* `e` *es un número. Por ejemplo,*

```
esNúm (Núm 3)                ⇝  True
esNúm (Sum (Núm 2) (Núm 3))  ⇝  False
```

Solución:

```
esNúm :: Expr -> Bool
esNúm (Núm _) = True
esNúm _       = False
```

Ejercicio 10.7.16. *Definir el generador*

```
arbExpDer' :: Gen (Expr,Expr)
```

que genere expresiones arbitrarias simplificadas junto con sus derivadas respecto de `x` *simplificadas. Por ejemplo,*

```
Main> muestra arbExpDer'
((-4)+y,0)
(x+x*z,1+z)
((-3)*y*z,0)
((-1),0)
((-3)+x,1)
```

Solución:

```
arbExpDer' :: Gen (Expr,Expr)
arbExpDer' =
  do e <- arbitrary
     return (simplifica e, simplifica (derivada e "x"))
```

Ejercicio 10.7.17. *Definir el tipo de datos* `XyzEnt` *para representar entornos con las variables* `x`*,* `y` *y* `z`*. Definir el generador*

```
xyzEnt :: Gen XyzEnt
```

de entornos. Por ejemplo,

```
Main> muestra xyzEnt
Xyz [("x",0),("y",0),("z",0)]
Xyz [("x",-1),("y",0),("z",0)]
Xyz [("x",0),("y",-2),("z",0)]
Xyz [("x",0),("y",0),("z",0)]
Xyz [("x",3),("y",-3),("z",-3)]
```

Hacer `XyzEnt` *subclase de* `Arbitrary` *con generador* `XyzEnt`*.*

Solución:

```
data XyzEnt = Xyz Ent
              deriving ( Show )

xyzEnt :: Gen XyzEnt
xyzEnt =
    do x <- arbitrary
       y <- arbitrary
       z <- arbitrary
       return (Xyz [("x",x),("y",y),("z",z)])

instance Arbitrary XyzEnt where
    arbitrary   = xyzEnt
    coarbitrary = undefined
```

Ejercicio 10.7.18. *Comprobar que la simplificación es correcta; es decir, el valor de una expresión es el mismo que el de su simplificada en cualquier entorno.*

Solución: La propiedad es

```
prop_SimplificaCorrecta e (Xyz ent) =
    valor ent e == valor ent (simplifica e)
```

La comprobación es

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

Ejercicio 10.7.19. *Comprobar si la simplificación elimina toda la basura; es decir, realiza todas las simplificaciones numéricas.*

Solución: La propiedad es

```
prop_SimplificaSinBasura e =
    sinBasura (simplifica e)
    where
      sinBasura (Sum a b) = not (esNúm a && esNúm b)
                            && sinBasura a && sinBasura b
      sinBasura (Pro a b) = not (esNúm a && esNúm b)
                            && sinBasura a && sinBasura b
      sinBasura _         = True
```

La comprobación es

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

Ejercicio 10.7.20. *Comprobar con QuickCheck si aes cierto que al derivar una expresión y simplificar su resultado se obtiene lo mismo que primero simplificarla y a continuación derivarla y simplificar su resultado.*

Solución: La propiedad es

```
prop_SimplificaDerivada e =
    simplifica (derivada e "x") == simplifica (derivada (simplifica e) "x")
```

La comprobación es

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

10.8. Árboles de enteros

Ejercicio 10.8.1. *Definir el tipo de datos* `Arbol` *para representar árboles binarios cuyos nodos son números enteros. Por ejemplo,*

```
Nodo 10 Vacio Vacio
Nodo 17 (Nodo 14 Vacio Vacio) (Nodo 9 Vacio Vacio)
```

Solución:

```
data Arbol = Vacio
           | Nodo Int Arbol Arbol
           deriving (Eq, Show)
```

Ejercicio 10.8.2. *Definir el generador*

```
arbArbol :: Int -> Gen Arbol
```

que genere árboles binarios de enteros arbitrarias. Por ejemplo,

```
Main> muestra (arbArbol 10)
Nodo (-1) (Nodo (-1) Vacio (Nodo 1 Vacio (Nodo 0 Vacio Vacio))) Vacio
Nodo 1 (Nodo 2 Vacio (Nodo 2 Vacio Vacio)) (Nodo 0 Vacio (Nodo 2 Vacio Vacio))
Nodo 0 Vacio (Nodo 0 Vacio (Nodo 0 Vacio Vacio))
Vacio
Nodo (-3) Vacio (Nodo (-2) Vacio (Nodo 5 Vacio (Nodo (-5) Vacio Vacio)))
```

Usar `arbArbol` *para hacer* `Arbol` *subclase de* `Arbitrary`.

Solución:

```
arbArbol :: Int -> Gen Arbol
arbArbol s =
    frequency [ (1, do return Vacio)
              , (s, do n <- arbitrary
                       a1 <- arbArbol s'
                       a2 <- arbArbol s'
                       return (Nodo n a1 a2))
              ]
    where
      s' = s `div` 2

instance Arbitrary Arbol where
    arbitrary   = sized arbArbol
    coarbitrary = undefined
```

Ejercicio 10.8.3. *Definir la función*

```
sumÁrbol :: Arbol -> Int
```

tal que `(sumaÁrbol a)` *es la suma de los nodos del árbol* `a`. *Por ejemplo,*

```
Main> sumaÁrbol (Nodo 17 (Nodo 14 Vacio Vacio) (Nodo 9 Vacio Vacio))
40
```

Solución:

```
sumaÁrbol :: Arbol -> Int
sumaÁrbol Vacio          = 0
sumaÁrbol (Nodo n a1 a2) = n + sumaÁrbol a1 + sumaÁrbol a2
```

Ejercicio 10.8.4. *Definir la función*

```
profundidad :: Arbol -> Int
```

tal que `(profundidad a)` *es la máxima profundidad de los nodos del árbol* `a`. *Por ejemplo,*

```
Main> profundidad (Nodo 17 (Nodo 14 Vacio Vacio) (Nodo 9 Vacio Vacio))
2
```

Solución:

```
profundidad :: Arbol -> Int
profundidad Vacio          = 0
profundidad (Nodo n a1 a2) = 1 + max (profundidad a1) (profundidad a2)
```

Ejercicio 10.8.5. *Definir la función*

```
ocurrencias :: Arbol -> Int -> Int
```

tal que `(ocurrencias a n)` *es el número de veces que aparece* `n` *como un nodo del árbol* `a`. *Por ejemplo,*

```
Main>
ocurrencias (Nodo 17 (Nodo 14 Vacio Vacio) (Nodo 17 Vacio Vacio)) 17
2
```

Solución:

```
ocurrencias :: Arbol -> Int -> Int
ocurrencias Vacio          _ = 0
ocurrencias (Nodo n a1 a2) p
    | n == p                 = 1 + ocurrencias a1 p + ocurrencias a2 p
    | otherwise              =     ocurrencias a1 p + ocurrencias a2 p
```

Ejercicio 10.8.6. *Definir la función*

```
izquierdo :: Arbol -> Arbol
```

tal que `(izquierdo a)` *es el subárbol izquierdo del árbol* `a`. *Por ejemplo,*

```
Main> izquierdo (Nodo 17 (Nodo 14 Vacio Vacio) (Nodo 17 Vacio Vacio))
Nodo 14 Vacio Vacio
```

Solución:

```
izquierdo :: Arbol -> Arbol
izquierdo (Nodo _ a _) = a
```

Ejercicio 10.8.7. *Definir la función*

```
derecho :: Arbol -> Arbol
```

tal que `(derecho a)` *es el subárbol derecho del árbol* `a`. *Por ejemplo,*

```
Main> derecho (Nodo 17 (Nodo 14 Vacio Vacio) (Nodo 17 Vacio Vacio))
Nodo 17 Vacio Vacio
```

Solución:

```
derecho :: Arbol -> Arbol
derecho (Nodo _ _ a) = a
```

Ejercicio 10.8.8. *Definir la función*

```
aplana :: Arbol -> [Int]
```

tal que `(aplana a)` *es la lista obtenida aplanando el árbol* `a`*. Por ejemplo,*

```
Main> aplana (Nodo 17 (Nodo 14 Vacio Vacio) (Nodo 29 Vacio Vacio))
[14,17,29]
```

Solución:

```
aplana :: Arbol -> [Int]
aplana Vacio          = []
aplana (Nodo x a1 a2) = aplana a1 ++ [x] ++ aplana a2
```

Ejercicio 10.8.9. *Comprobar con QuickCheck que la suma de los elementos de la lista obtenida aplanando un árbol* `a` *es igual que la suma de los nodos de* `a`*.*

Solución: La propiedad es

```
prop_SumaAplana a =
    sum (aplana a) == sumaÁrbol a
```

La comprobación es

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

Ejercicio 10.8.10. *Definir la función*

```
máximoA :: Arbol -> Int
```

tal que `(máximo a)` *es el máximo de los nodos del árbol* `a`*. Por ejemplo,*

```
Main> máximoA (Nodo 17 (Nodo 14 Vacio Vacio) (Nodo 29 Vacio Vacio))
29
```

Solución:

```
máximoA :: Arbol -> Int
máximoA a = maximum (aplana a)
```

Ejercicio 10.8.11. *Definir la función*

```
espejo :: Arbol -> Arbol
```

tal que `(espejo a)` *es la imagen especular del árbol* `a`*. Por ejemplo,*

```
Main> espejo (Nodo 17 (Nodo 14 Vacio Vacio) (Nodo 29 Vacio Vacio))
Nodo 17 (Nodo 29 Vacio Vacio) (Nodo 14 Vacio Vacio)
```

Solución:

```
espejo :: Arbol -> Arbol
espejo Vacio          = Vacio
espejo (Nodo n a1 a2) = Nodo n (espejo a2) (espejo a1)
```

Ejercicio 10.8.12. *Comprobar con QuickCheck que la imagen especular de la imagen especular de un árbol es el propio árbol.*

Solución: La propiedad es

```
prop_EspejoEspejo a =
    espejo (espejo a) == a
```

La comprobación es

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

Ejercicio 10.8.13. *Comprobar con QuickCheck al aplanar la imagen especular de un árbol* `a` *se obtiene lo mismo que al invertir la lista obtenida aplanando* `a`*.*

Solución:

```
prop_AplanaEspejo a =
    aplana (espejo a) == reverse (aplana a)
```

La comprobación es

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

Capítulo 11

Analizadores

Contenido

Esta capítulo está basada en el libro de B.C. Ruiz y otros "Razonando con Haskell".

11.1. Analizadores mediante listas de comprensión

Ejercicio 11.1.1. *Calcular el valor de las siguientes expresiones*

```
read "123" :: Int
read "True" :: Bool
read "True" :: Int
read "[1,2,3]" :: [Int]
read "(1,False,1.5)" :: (Int,Bool,Float)
```

Determinar el tipo de la función `read`.

Solución: Loa valores son

```
Main> read "123" :: Int
123
Main> read "True" :: Bool
True
Main> read "True" :: Int
Program error: Prelude.read: no parse
Main> read "[1,2,3]" :: [Int]
[1,2,3]
Main> read "(1,False,1.5)" :: (Int,Bool,Float)
(1,False,1.5)
```

Para determinar el tipo de `read` se ejecuta la siguiente orden

```
Main> :t read
read :: (Read a) => String -> a
```

Nota: Un **analizador** para un tipo *a* es una función que toma como argumento una cadena y devuelve una lista de pares formado por un elemento de tipo *a* y una cadena. El tipo de los analizadores para *a* está definido por

```
type ReadS a = String -> [(a,String)]
```

Ejercicio 11.1.2. *Definir el analizador*

```
exito :: a -> ReadS a
```

tal que `(exito x c)` *no consume nada de la cadena de entrada* `c` *y devuelve siempre una única solución (el valor* `x`*). Por ejemplo,*

```
exito 'X' "123hola"  ⇝  [('X',"123hola")]
exito 'z' "23"       ⇝  [('z',"23")]
```

Solución:

```
exito :: a -> ReadS a
exito x = \c -> [(x,c)]
```

Ejercicio 11.1.3. *Definir el analizador*

```
epsilon :: ReadS ()
```

tal que `(epsilon c)` *no consume nada de la cadena de entrada* `c` *y devuelve siempre una única solución (el valor* `()`*). Por ejemplo,*

```
epsilon "123hola"  ⇝  [((),"123hola")]
```

Solución:

```
epsilon :: ReadS ()
epsilon = exito ()
```

Ejercicio 11.1.4. *Definir el analizador*

```
fallo :: ReadS a
```

tal que `(fallo c)` *siempre falla; es decir, siempre devuelve la lista vacía.*

Solución:

```
fallo :: ReadS a
fallo = \c -> []
```

Ejercicio 11.1.5. *Definir el analizador*

```
rChar :: Char -> ReadS Char
```

tal que `(rChar c s)` *tiene éxito si* `c` *es el primer carácter de la cadena de* `s`*. En caso de éxito consume el primer carácter de* `s`*. Por ejemplo,*

```
rChar 'a' "abc"  ⇝  [('a',"bc")]
rChar 'b' "abc"  ⇝  []
```

Solución:

```
rChar :: Char -> ReadS Char
rChar c = \s -> case s of
                  []   -> []
                  x:xs -> if c==x then [(x,xs)] else []
```

Ejercicio 11.1.6. *Definir el analizador*

```
rSat :: (Char -> Bool) -> ReadS Char
```

tal que `(rSat p s)` *tiene éxito si el primer carácter de la cadena de* `s` *satisface el predicado* `p`*. En caso de éxito consume el primer carácter de* `s`*. Por ejemplo,*

```
rSat isUpper "ABC"  ⇝  [('A',"BC")]
rSat isLower "ABC"  ⇝  []
```

Solución:

```
rSat :: (Char -> Bool) -> ReadS Char
rSat p = \s -> case s of
                  []   -> []
                  x:xs -> if p x then [(x,xs)] else []
```

Ejercicio 11.1.7. *Definir el analizador*

```
rChar' :: Char -> ReadS Char
```

equivalente a `rChar`*, usando* `rSat`*.*

Solución:

```
rChar' :: Char -> ReadS Char
rChar' x = rSat (== x)
```

Ejercicio 11.1.8. *Definir el combinador*

```
(-+-) :: ReadS a -> ReadS a -> ReadS a
```

tal que `(p1 -+- p2)` *tiene éxito si lo tiene el analizador* `p1` *o el* `p2`*. En caso de éxito, develve los resultados de aplicar* `p1` *más los de aplicar* `p2`*. Por ejemplo,*

```
(rChar 'a' -+- rChar 'b') "abc"     ⇝  [('a',"bc")]
(rChar 'a' -+- rChar 'b') "bca"     ⇝  [('b',"ca")]
(rChar 'a' -+- rSat isLower) "abc"  ⇝  [('a',"bc"),('a',"bc")]
```

Solución:

```
infixl 5 -+-
(-+-) :: ReadS a -> ReadS a -> ReadS a
p1 -+- p2 = \s -> p1 s ++ p2 s
```

Ejercicio 11.1.9. *Definir el analizador*

```
(>=>) :: ReadS a -> (a -> b) -> ReadS b
```

tal que `((p >=> f) s)` *tiene éxito si lo tiene* `(p s)`*. En caso de éxito devuelve los resultados de aplicar* `f` *a las primeras componentes de* `(p s)`*. Por ejemplo,*

```
(rSat isUpper >=> ord) "ABC"  ⇝  [(65,"BC")]
```

Solución:

```
infixl 6 >=>
(>=>) :: ReadS a -> (a -> b) -> ReadS b
p >=> f = \s -> [(f x,s1) | (x,s1) <- p s]
```

Ejercicio 11.1.10. *Definir el analizador*

```
(&><) :: ReadS a -> ReadS b -> ReadS (a,b)
```

tal que `((p1 &>< p2) s)` *tiene éxito si* `p1` *reconoce el primer carácter de* `s` *y* `p2` *el segundo. En caso de éxito devuelve el par formado por los dos primeros caracteres de* `s`*. Por ejemplo,*

```
(rChar 'a' &>< rChar 'b') "abcd"               ⇝  [(('a','b'),"cd")]
(rChar 'a' &>< rChar 'c') "abcd"               ⇝  []
(rChar 'a' &>< rChar 'b' &>< rChar 'c') "abcd"  ⇝  [((('a','b'),'c'),"d")]
```

Solución:

```
infixl 7 &><
(&><) :: ReadS a -> ReadS b -> ReadS (a,b)
p1 &>< p2 = \s -> [((x1,x2),s2) | (x1,s1) <- p1 s,
                                  (x2,s2) <- p2 s1]
```

Ejercicio 11.1.11. *Definir el analizador*

```
(>>>) :: ReadS a -> (a -> ReadS b) -> ReadS b
```

tal que `((p >>> f) s)` *tiene éxito si* `(p s)` *tiene éxito con valor* `(x,s1)` *y* `((f x) s1` *tiene éxito con valor* `(y,s2)`*. En caso de éxito devuelve* `(y,s2)`*. Para ejmplo, consultar los dos siguientes ejercicios.*

Solución:

```
infixr 6 >>>
(>>>) :: ReadS a -> (a -> ReadS b) -> ReadS b
p >>> f = \s -> [ (y,s2) | (x,s1)  <-  p s,
                           (y,s2)  <-  (f x) s1 ]
```

Ejercicio 11.1.12. *Definir el analizador*

```
rAB :: ReadS (Char, Char)
```

tal que `(rAB s)` *tiene éxito si los dos primeros caracteres de* `s` *son letras mayúsculas. En caso de éxito, devuelve el par formado por los dos primeros caracteres de* `s`*. Por ejemplo,*

```
rAB "ABCde"  ⇝  [(('A','B'),"Cde")]
rAB "AbCde"  ⇝  []
```

Solución:

```
rAB :: ReadS (Char, Char)
rAB = rSat isUpper >>> (\x ->
      rSat isUpper >>> (\y ->
      exito (x,y)))
```

Ejercicio 11.1.13. *Definir el analizador*

```
rDosIguales :: ReadS (Char, Char)
```

tal que `(rDosIguales s)` *tiene éxito si los dos primeros caracteres de* `s` *son dos letras mayúsculas iguales. En caso de éxito, devuelve el par formado por los dos primeros caracteres de* `s`*. Por ejemplo,*

```
rDosIguales "AAbcd"  ⇝  [(('A','A'),"bcd")]
rDosIguales "AEbcd"  ⇝  []
rDosIguales "Aabcd"  ⇝  []
rDosIguales "aabcd"  ⇝  []
```

Solución:

```
rDosIguales :: ReadS (Char, Char)
rDosIguales = rSat isUpper >>> \x ->
              rChar x      >>> \y ->
              exito (x,y)
```

Ejercicio 11.1.14. *Definir el analizador*

```
rep1 :: ReadS a -> ReadS [a]
```

tal que `((rep1 p) s)` *aplica el analizador* `p` *una o más veces a* `s` *y devuelve el resultado en una lista. Por ejemplo,*

```
(rep1 (rChar 'a')) "aabc"     ⇝  [("aa","bc"),("a","abc")]
(rep1 (rSat isDigit)) "12ab"  ⇝  [("12","ab"),("1","2ab")]
```

Solución:

```
rep1 :: ReadS a -> ReadS [a]
rep1 p = varios -+- uno
    where
      varios = p            >>> \x  ->
               rep1 p       >>> \xs ->
               exito (x:xs)
      uno    = p            >>> \x  ->
               exito [x]
```

Ejercicio 11.1.15. *Definir el analizador*

```
rep0 :: ReadS a -> ReadS [a]
```

tal que `((rep0 p) s)` *aplica el analizador* `p` *cero o más veces a* `s` *y devuelve el resultado en una lista. Por ejemplo,*

```
(rep0 (rChar 'a')) "aabc"     ⇝  [("aa","bc"),("a","abc"),("","aabc")]
(rep0 (rSat isDigit)) "12ab"  ⇝  [("12","ab"),("1","2ab"),("","12ab")]
```

Solución:

```
rep0 :: ReadS a -> ReadS [a]
rep0 p = rep1 p
         -+-
         epsilon  >>> \() ->
         exito []
```

Ejercicio 11.1.16. *Definir el analizador*

```
rNumNatural' :: ReadS Integer
```

tal que `(rNumNatural' s)` *tiene éxito si los primeros caracteres de* `s` *son números naturales. Por ejemplo,*

```
rNumNatural' "12ab"  ⇝  [(12,"ab"),(1,"2ab")]
```

Solución:

```
rNumNatural' :: ReadS Integer
rNumNatural' = rep1 (rSat isDigit) >>> \cs ->
               exito (aInteger cs)
    where
      chrAInteger  :: Char -> Integer
      chrAInteger c = toInteger (ord c - ord '0')
      aInteger     :: [Char] -> Integer
      aInteger      = foldl1 (\x y -> 10*x + y) . map chrAInteger
```

Ejercicio 11.1.17. *Definir el analizador*

```
rNumNatural :: ReadS Integer
```

tal que `(rNumNatural s)` *tiene éxito si los primeros caracteres de* `s` *son números naturales. En caso de éxito, devuelve el mayor prefijo de* `s` *formado por números naturales. Por ejemplo,*

```
rNumNatural "12ab"   ⇝  [(12,"ab")]
rNumNatural "x12ab"  ⇝  []
```

Solución: La definición es

```
rNumNatural :: ReadS Integer
rNumNatural = primero rNumNatural'
```

donde `((primero p) s)` es el primer resultado de `(p s)`. Por ejemplo,

```
primero rNumNatural' "12ab"  ⇝  [(12,"ab")]
primero rNumNatural' "a12ab"  ⇝  []
```

La definición de `primero` es

```
primero :: ReadS a -> ReadS a
primero p = \s -> case p s of
                    []  ->  []
                    x:_ ->  [x]
```

Ejercicio 11.1.18. *Definir el analizador*

```
(?) :: ReadS a -> a -> ReadS a
```

tal que `((p ? e) s)` *devuelve el resultado de* `(p s)` *si* `(p s)` *tiene éxisto y* `e` *en caso contrario. Por ejemplo,*

```
((rSat isDigit) ? 'f') "12ab"   ⇝  [('1',"2ab"),('f',"12ab")]
((rSat isDigit) ? 'f') "x12ab"  ⇝  [('f',"x12ab")]
```

Solución:

```
infix 8 ?
(?) :: ReadS a -> a -> ReadS a
p ? ifNone = p
             -+-
             epsilon >>> \() ->
             exito ifNone
```

Ejercicio 11.1.19. *Definir el analizador*

```
rNumEntero :: ReadS Integer
```

tal que `(rNumEntero s)` *tiene éxito si el primer carácter de* `s` *es un signo (*`+` *ó* `-`*, por defecto es* `+`*) y los siguientes caracteres de* `s` *son números naturales. Por ejemplo,*

```
rNumEntero "12ab"   ⇝  [(12,"ab")]
rNumEntero "+12ab"  ⇝  [(12,"ab")]
rNumEntero "-12ab"  ⇝  [(-12,"ab")]
```

Solución:

```
rNumEntero :: ReadS Integer
rNumEntero = (rChar '+' -+- rChar '-') ? '+' >>> \s ->
              rNumNatural                    >>> \n ->
              exito (aNúmero s n)
    where
      aNúmero '+' n =   n
      aNúmero '-' n =  -n
```

Ejercicio 11.1.20. *Definir el analizador*

```
rListaEnteros' :: ReadS [Integer]
```

tal que `(rListaEnteros' s)` *tiene éxito si* `s` *comienza por una lista de números enteros. Por ejemplo,*

```
rListaEnteros' "[1,-2]ab"  ⇝  [([1,-2],"ab")]
```

Solución:

```
rListaEnteros' :: ReadS [Integer]
rListaEnteros' =
    rChar '[' >>> \_  ->
    rElems    >>> \es ->
    rChar ']' >>> \_  ->
    exito es
    where
      rElems      = rNumEntero       >>> \n  ->
                    rep0 rComaEntero >>> \ns ->
                    exito (n:ns)
      rComaEntero = rChar ','        >>> \_ ->
                    rNumEntero       >>> \n ->
                    exito n
```

Ejercicio 11.1.21.

Ejercicio 11.1.22. *Definir el analizador*

```
rString :: String -> ReadS String
```

tal que `((rString s1) s2)` *tiene éxito si* `s1` *es un prefijo de* `s2`*. Por ejemplo,*

```
rString "ab" "ab xy"     ⇝  [("ab"," xy")]
rString "ab" "abcd xy"   ⇝  [("ab","cd xy")]
rString "ab" "12 ab xy"  ⇝  []
```

Solución:

```
rString :: String -> ReadS String
rString []     = exito ""
rString (c:cs) = rChar   c  >>> \_  ->
                 rString cs >>> \rs ->
                 exito (c:rs)
```

Ejercicio 11.1.23. *Definir el analizador*

```
rLex :: ReadS a -> ReadS a
```

tal que `((rLex p) s)` *tiene éxito si* `s` *es una cadena de espacios en blanco seguida de una cadena* `s1` *y* `(p s1)` *tiene éxito. Por ejemplo,*

```
rSat isDigit "   12ab"         ⇝  []
rLex (rSat isDigit) "   12ab"  ⇝  [('1',"2ab")]
```

Solución:

```
rLex :: ReadS a -> ReadS a
rLex p = rep0 (rSat esEspacio) >>> \_ ->
         p
    where
      esEspacio = (`elem` " \n\t")
```

Ejercicio 11.1.24. *Definir el analizador*

```
rToken :: String -> ReadS String
```

tal que `((rToken s1) s2)` *tiene éxito si* `s1` *es un prefijo de* `s2` *sin considerar los espacios en blanco iniciales de* `s2`*. Por ejemplo,*

```
rToken "[" "   [1,2]ab"  ⇝  [("[","1,2]ab")]
```

Solución:

```
rToken :: String -> ReadS String
rToken s = rLex (rString s)
```

Ejercicio 11.1.25. *Definir el analizador*

```
rNat :: ReadS Integer
```

tal que `(rNat s)` *tiene éxito si los primeros caracteres de* `s` *son números naturales sin considerar los espacios en blanco iniciales de* `s`*. Por ejemplo,*

```
rNat "  12ab"  ⇝  [(12,"ab")]
```

Solución:

```
rNat :: ReadS Integer
rNat = rLex rNumNatural
```

Ejercicio 11.1.26. *Definir el analizador*

```
rInteger :: ReadS Integer
```

tal que `(rInteger s)` *tiene éxito si el primer carácter de* `s` *es un signo (+ ó -, por defecto es +) y los siguientes caracteres de* `s` *son números naturales, sin considerar los espacios en blanco iniciales de* `s`*. Por ejemplo,*

```
rInteger "  12ab"  ⇝  [(12,"ab")]
rInteger " -12ab"  ⇝  [(-12,"ab")]
rInteger " +12ab"  ⇝  [(12,"ab")]
```

Solución:

```
rInteger :: ReadS Integer
rInteger = rLex rNumEntero
```

Ejercicio 11.1.27. *Definir el analizador*

```
rListaEnteros :: ReadS [Integer]
```

tal que `(rListaEnteros s)` *tiene éxito si* `s` *comienza por una lista de números enteros sin considerar los posibles espacios en blanco. Por ejemplo,*

```
rListaEnteros "[ 1 , -2,  3]ab"  ⇝  [([1,-2,3],"ab")]
```

Solución:

```
rListaEnteros :: ReadS [Integer]
rListaEnteros =
    rToken "["  >>> \_  ->
    rElems      >>> \es ->
    rToken "]"  >>> \_  ->
    exito es
    where
      rElems      = (rInteger        >>> \n  ->
                     rep0 rComaEntero >>> \ns ->
                     exito (n:ns))
                     -+-
                     (epsilon        >>> \() ->
                     exito [])
      rComaEntero = rToken ","       >>> \_ ->
                     rInteger        >>> \n ->
                     exito n
```

Ejercicio 11.1.28. *Evaluar las siguientes expresiones*

```
(reads :: ReadS Int) "12ab"
(reads "12 hola") :: [(Integer,String)]
```

Solución: La evaluación es

```
Main> (reads :: ReadS Int) "12ab"
[(12,"ab")]
Main> (reads "12 hola") :: [(Integer,String)]
[(12," hola")]
```

Ejercicio 11.1.29. *Definir el tipo de datos* `Arbol` *para representar los árboles binarios. Por ejemplo,*

```
Nodo (Hoja 1) (Nodo (Hoja 2) (Hoja 3))
```

representa un árbol binario cuya rama izquierda es la hoja 1 y la rama derecha es un árbol binario con rama izquierda la hoja 2 y rama derecha la hoja 3.

Solución:

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

Ejercicio 11.1.30. *Definir el ejemplo del árbol del ejercicio anterior como* `ejArbol`*.*

Solución:

```
ejArbol :: Arbol Int
ejArbol = Nodo (Hoja 1) (Nodo (Hoja 2) (Hoja 3))
```

Ejercicio 11.1.31. *Definir la función*

```
muestraArbol :: (Show a) => Arbol a -> String
```

para escribir los árboles. Por ejemplo,

```
muestraArbol ejArbol  ⇝  "<1|<2|3>>"
```

Solución:

```
muestraArbol :: (Show a) => Arbol a -> String
muestraArbol x = muestraArbol' x ""
```

donde `(muestraArbol' a s)` es la cadena obtenida añadiendo la representación del árbol `a` a la cadena `s`. Por ejemplo,

```
muestraArbol' ejArbol "abcd"  ⇝  "<1|<2|3>>abcd"
```

La definición de `muestraArbol'` es

```
muestraArbol' :: (Show a) => Arbol a -> ShowS
muestraArbol' (Hoja x)
    =  shows x
muestraArbol' (Nodo l r)
    =  ('<':) . muestraArbol' l . ('|':) . muestraArbol' r . ('>':)
```

Ejercicio 11.1.32. *Hacer la clase de los árboles una instancia de la de los objetos imprimibles, usando* `muestraArbol` *como función de escritura. Por ejemplo,*

```
Main> Nodo (Hoja 1) (Nodo (Hoja 2) (Hoja 3))
<1|<2|3>>
```

Solución:

```
instance Show a => Show (Arbol a) where
    show = muestraArbol
```

Ejercicio 11.1. *Definir el analizador*

```
leeArbol :: (Read a) => ReadS (Arbol a)
```

tal que `(leeArbol s)` *tiene éxito si el prefijo de* `s` *es un árbol. Por ejemplo,*

```
Main> leeArbol "<1|<2|3>>" :: [(Arbol Int,String)]
[(<1|<2|3>>,"")]
Main> leeArbol "<1|<2|3>>abcd" :: [(Arbol Int,String)]
[(<1|<2|3>>,"abcd")]
```

Solución:

```
leeArbol :: (Read a) => ReadS (Arbol a)
leeArbol ('<':s) =  [(Nodo l r, u) | (l, '|':t) <- leeArbol s,
                                     (r, '>':u) <- leeArbol t ]
leeArbol s       =  [(Hoja x, t) | (x,t) <- reads s]
```

Ejercicio 11.1.33. *Hacer la clase de los árboles una instancia de la de los objetos legibles, usando* `leeArbol` *como función de lectura. Por ejemplo,*

```
Main> reads "<1|<2|3>>" :: [(Arbol Int,String)]
[(<1|<2|3>>,"")]
Main> reads "<1 | <2|3>>" :: [(Arbol Int,String)]
[]
Main> read "<1|<2|3>>" :: Arbol Int
<1|<2|3>>
```

Solución:

```
instance Read a => Read (Arbol a) where
    readsPrec _ s = leeArbol s
```

Parte III

Programación avanzada y aplicaciones

Capítulo 12

Búsqueda en grafos y espacios de estados

Contenido

12.1. Búsqueda en profundidad en grafos

Ejercicio 12.1. *Definir el módulo de búsqueda en grafos como* `Bus_prof_en_grafos`

Solución:

```
module Busq_prof_en_grafos where
```

Ejercicio 12.2. *Un grafo de tipo* `v` *es un tipo de datos compuesto por la lista de vértices y la función que asigna a cada vértice la lista de sus sucesores. Definir el tipo* `Grafo`.

Solución:

```
data Grafo v = G [v] (v -> [v])
```

Nota. El grafo de la figura 12.1 se representa por

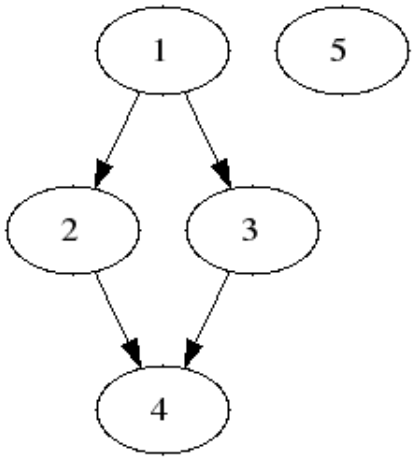


Figura 12.1: Ejemplo de grafo

```
ej_grafo = G [1..5] suc
    where suc 1 = [2,3]
          suc 2 = [4]
          suc 3 = [4]
          suc _ = []
```

Nota. `camino g u v` es un camino (i.e una lista de vértices tales que cada uno es un sucesor del anterior) en el grafo `g` desde el vértice `u` al `v`. Por ejemplo,

```
camino ej_grafo 1 4 ⇝ [4,2,1]
```

```
camino :: Eq a => Grafo a -> a -> a -> [a]
camino g u v = head (caminosDesde g u (== v) [])
```

Nota. `caminosDesde g o te vis` es la lista de los caminos en el grafo `g` desde el vértice origen `o` hasta vértices finales (i.e los que verifican el test de encontrado `te`) sin volver a pasar por los vértices visitados `vis`. Por ejemplo,

```
caminosDesde ej_grafo 1 (==4) [] ⇝ [[4,2,1],[4,3,1]]
```

```
caminosDesde :: Eq a => Grafo a -> a -> (a -> Bool) -> [a] -> [[a]]
caminosDesde g o te vis
    | te o      = [o:vis]
    | otherwise = concat [caminosDesde g o' te (o:vis)
                          | o' <- suc o,
                            notElem o' vis]
    where G _ suc = g
```

Nota. Para hacer la búsqueda en anchura basta cambiar en la definición de `caminosDesde` la expresión `o:vis` por `vis++[o]`.

12.2. Búsqueda en profundidad en grafos con pesos

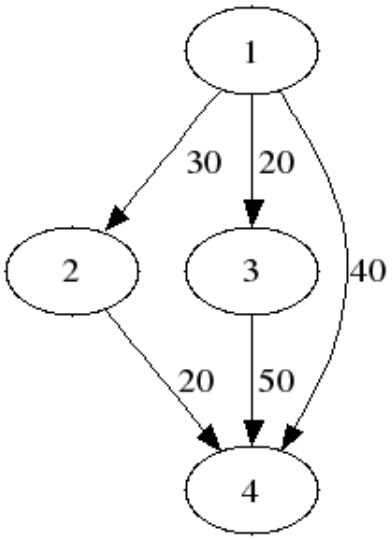
Nota. El módulo de búsqueda en grafos es `Busq_prof_en_grafos_con_pesos`

```
module Busq_prof_en_grafos_con_pesos where
```

Nota. Un grafo con pesos de tipo `v` es un tipo de datos compuesto por la lista de vértices y la función que asigna a cada vértice la lista de sus sucesores junto con el coste de la transición.

```
data Grafo v p = G [v] (v -> [(v,p)])
```

Nota. El grafo



se representa por

```
ej_grafo = G [1..4] suc
    where suc 1 = [(2,30), (3,20), (4,40)]
          suc 2 = [(4,20)]
          suc 3 = [(4,50)]
          suc _ = []
```

Nota. `caminos g u v p` es la lista de caminos en el grafo `g` desde el vértice `u` al `v` de coste menor o igual que `p`. Por ejemplo,

```
caminos ej_grafo 1 4 40 ⇝ [[4,1]]
caminos ej_grafo 1 4 50 ⇝ [[4,2,1],[4,1]]
caminos ej_grafo 1 4 70 ⇝ [[4,2,1],[4,3,1],[4,1]]
```

```
caminos :: (Eq a, Num b, Ord b) => Grafo a b -> a -> a -> b -> [[a]]
caminos g u v pt = caminosDesde g u (\x _ -> x==v) (> pt) [] 0
```

Nota. `caminosDesde g o te tr vis p` es la lista de los caminos en el grafo `g` desde el vértice origen `o` hasta vértices finales (i.e los que verifican el test de encontrado `te`) que no están podados por el test de retorno `tr` teniendo en cuenta los vértices visitados `vis` y el peso consumido `p`. Por ejemplo,

```
caminosDesde ej_grafo 1 (\x _ -> x==4) (> 50) [] 0 \valor [[4,2,1],[4,1]]
```

```
caminosDesde :: (Eq a, Num b) => Grafo a b -> a -> (a -> b -> Bool) ->
                                 (b -> Bool) -> [a] -> b -> [[a]]
caminosDesde g o te tr vis p
    | te o p    = [o:vis]
    | otherwise = concat [caminosDesde g o' te tr (o:vis) np |
                          (o',p') <- suc o,
                          notElem o' vis,
                          let np = p+p',
                          not(tr np)]
    where G _ suc = g
```

Nota. `costeCamino g c` es el coste del camino c en el grafo g. Por ejemplo,

```
costeCamino ej_grafo [4,1]   ⇝ 40
costeCamino ej_grafo [4,2,1] ⇝ 50
costeCamino ej_grafo [4,3,1] ⇝ 70
```

```
costeCamino :: (Num a, Eq b) => Grafo b a -> [b] -> a
costeCamino _ [_]      = 0
costeCamino g (u:v:xs) = p + costeCamino g (v:xs)
                         where G _ suc = g
                               Just p  = lookup u (suc v)
```

Nota. `insertaCamino g c l` inserta el camino `c` del grafo `g` en la lista de caminos `l` delante del primer elemento de `l` de coste mayor o igual que `e`. Por ejemplo,

```
insertaCamino ej_grafo [4,2,1] [[4,1],[4,3,1]] ⇝ [[4,1],[4,2,1],[4,3,1]]
```

```
insertaCamino :: (Num a, Eq b, Ord a) => Grafo b a -> [b] -> [[b]] -> [[b]]
insertaCamino g c []     = [c]
insertaCamino g c (x:xs)
    | costeCamino g c <= costeCamino g x = c:x:xs
    | otherwise                          = x : insertaCamino g c xs
```

Es equivalente a la función `insert` definida en `Data.List`

Nota. `ordenaCaminos l` es la lista `l` ordenada mediante inserción, Por ejemplo,

```
ordenaCaminos ej_grafo [[4,2,1],[4,3,1],[4,1]] ⇝ [[4,1],[4,2,1],[4,3,1]]
```

```
ordenaCaminos :: (Ord a, Eq b, Num a) => Grafo b a -> [[b]] -> [[b]]
ordenaCaminos g  = foldr (insertaCamino g) []
```

Nota. `mejorCamino g u v` es el camino de menor coste en el grafo `g` desde `u` hasta `v`. Por ejemplo.

```
mejorCamino ej_grafo 1 4 ⇝ [4,1]
```

```
mejorCamino :: (Eq a, Num b, Ord b) => Grafo a b -> a -> a -> [a]
mejorCamino g u v = head(ordenaCaminos g (caminosDesde g
                                                       u
                                                       (\x _ -> x==v)
                                                       (\_ -> False)
                                                       []
                                                       0))
```

Puden suprimirse los paréntesis usando el operador `$`

```
mejorCamino' :: (Eq a, Num b, Ord b) => Grafo a b -> a -> a -> [a]
mejorCamino' g u v = head
                     $ ordenaCaminos g
                     $ caminosDesde g
                                    u
                                    (\x _ -> x==v)
                                    (\_ -> False)
                                    []
                                    0
```

Nota. Se puede redefinir `mejorCamino` de forma que en `caminosDesde` se inserte el nuevo camino en la lista ordenada de caminos pendientes de expandir como en Prolog.

12.3. La clase grafo con búsqueda en profundidad

Nota. El módulo de grafos con búsqueda en profundidad es `Grafo_busq_prof`

```
module Grafo_busq_prof where
```

Nota. La clase de los grafos con búsqueda en profundidad de tipo `v` consta de los siguientes métodos primitivos:

1. `vértices` es la lista de los vértices del grafo.
2. `suc x` es la lista de los sucesores del vértice `x`.

y de los siguientes métodos definidos:

1. `camino u v` es un camino (i.e una lista de vértices tales que cada uno es un sucesor del anterior) desde el vértice `u` al `v`.
2. `caminosDesde o te vis` es la lista de los caminos desde el vértice origen `o` hasta vértices finales (i.e los que verifican el test de encontrado `te`) a partir de los vértices visitados `vis`.
3. `tv x ys` se verifica si el vértice `x` cumple el test de visitabilidad respecto de la lista de vértices `ys`. El test de visitabilidad por defecto es que `x` no pertenezca a `ys` (para evitar ciclos en el grafo).

Para definir un grafo basta determinar los vértices y sucesores. Los tres restantes métodos están definidos a partir de ellos.

```
class Eq v => Grafo v where
  vértices     :: [v]
  suc          :: v -> [v]
  camino       :: v -> v -> [v]
  caminosDesde :: v -> (v -> Bool) -> [v] -> [[v]]
  tv           :: v -> [v] -> Bool
  -- Métodos por defecto:
  camino u v = head (caminosDesde u (== v) [])

  caminosDesde o te vis
      | te o      = [o:vis]
      | otherwise = concat [caminosDesde o' te (o:vis) |
                            o' <- suc o,
                            tv o' vis]

  tv = notElem
```

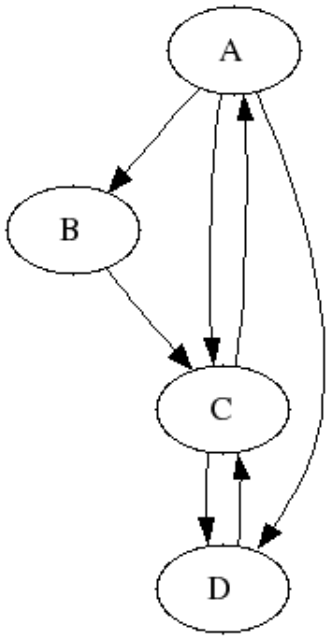


Figura 12.2: Grafo 1

Nota. El grafo de la figura se representa por

```
module Grafo_busq_prof_ej_1 where
import Grafo_busq_prof

data Vértice = A | B | C | D | E deriving (Show,Eq,Enum)

instance Grafo Vértice where
    vértices = [A .. E]
    suc A = [B,C,D]
    suc B = [C]
    suc C = [A,D]
    suc D = [C]
    suc E = []
```

Con los métodos definidos podemos calcular

```
camino A D                 ⇝ [D,C,B,A]
caminosDesde A (== D) []   ⇝ [[D,C,B,A],[D,C,A],[D,A]]
tv B [A,B,C]               ⇝ False
tv D [A,B,C]               ⇝ True
```

Nota. El grafo de la figura 12.2, donde el vértice `B` no es visitable se representa por

```
module Grafo_busq_prof_ej_2 where
import Grafo_busq_prof

data Vértice = A | B | C | D | E deriving (Show,Eq,Enum)

instance Grafo Vértice where
    vértices = [A .. E]
    suc A = [B,C,D]
    suc B = [C]
    suc C = [A,D]
    suc D = [C]
    suc E = []
    tv x ys = notElem x ys && x /= B
```

Con los métodos definidos podemos calcular

```
camino A D                ⇝ [D,C,A]
caminosDesde A (== D) []  ⇝ [[D,C,A],[D,A]]
tv B [A,B,D]              ⇝ False
tv D [A,B,C]              ⇝ True
tv B [A,C,E]              ⇝ False
```

Nota. El grafo de la figura 12.2, donde cada vértice es visitable hasta dos veces se representa por

```
module Grafo_busq_prof_ej_3 where
import Grafo_busq_prof
import Data.List ((\\))

data Vértice = A | B | C | D | E deriving (Show,Eq,Enum)

instance Grafo Vértice where
    vértices = [A .. E]
    suc A = [B,C,D]
    suc B = [C]
    suc C = [A,D]
    suc D = [C]
    suc E = []
    tv x ys = notElem x (ys \\ [x])
```

Con los métodos definidos podemos calcular

```
camino A D                ⇝ [D,C,B,A,C,B,A]
caminosDesde A (== D) [] ⇝ [[D,C,B,A,C,B,A],
                                [D,C,A,C,B,A],
                                [D,A,C,B,A],
                                [D,C,B,A],
                                [D,C,B,A,C,A],
                                [D,C,A,C,A],
                                [D,A,C,A],
                                [D,C,A],
                                [D,A]]
tv B [A,B,D]              ⇝ True
tv B [A,B,D,B]            ⇝ False
tv D [A,B,C]              ⇝ True
```

12.4. La clase grafo con búsqueda con pesos

Nota. El módulo de grafos con búsqueda con pesos es `Grafo_busq_con_pesos`

```
module Grafo_busq_con_pesos where
```

Nota. Un `Arco` es un par formado por un vértice y un peso.

```
data Arco  v p = Arc v p
type Arcos v p = [Arco v p]
```

Nota. La clase de los grafos con búsqueda con pesos de tipo `v` consta de los siguientes métodos primitivos:

1. `vértices` es la lista de los vértices del grafo.
2. `suc x` es la lista de los sucesores del vértice `x`, donde cada sucesor de un vérrtice `x` es un `Arco y p` donde `y` es el vértice sucesor de `x` y `p` es el coste de ir de `x` a `y`.

y de los siguientes métodos definidos:

1. `camino u v p` es la lista de caminos desde el vértice `u` al `v` de coste menor o igual que `p`.
2. `caminosDesde o te tr vis p` es la lista de los caminos desde el vértice origen `o` hasta vértices finales (i.e los que verifican el test de encontrado `te`) que no están podados por el test de retorno `tr` teniendo en cuenta los vértices visitados `vis` y el peso consumido `p`.

3. `tv a ys` se verifica si el arco `a` cumple el test de visitabilidad respecto de la lista de vértices `ys`. El test de visitabilidad por defecto es que el vértice de `a` no pertenezca a `ys` (para evitar ciclos en el grafo).

Para definir un grafo basta determinar los vértices y sucesores. Los tres restantes métodos están definidos a partir de ellos.

```
class Eq v => GrafoPesos v where
  vértices     :: [v]
  suc          :: Num p => v -> Arcos v p
  caminos      :: (Num p,Ord p) => v -> v -> p -> [[v]]
  caminosDesde :: Num p => v -> (v->p->Bool) -> (p->Bool) ->
                           [v] -> p -> [[v]]
  tv           :: Num p => (v,p) -> [v] -> Bool
  -- Métodos por defecto:
  caminos u v pt = caminosDesde u (\x _ -> x==v) (> pt) [] 0

  tv (x,_) ys = notElem x ys

  caminosDesde o te tr vis p
      | te o p    = [o:vis]
      | otherwise = concat [caminosDesde o' te tr (o:vis) np |
                            Arc o' p' <- suc o,
                            tv (o',p') vis,
                            let np = p+p',
                            not(tr np)]
```

Nota. El grafo de la figura 12.3 se representa por

```
module Grafo_busq_con_pesos_ej_1 where
import Grafo_busq_con_pesos

instance GrafoPesos Int where
   suc 1 = [Arc 2 30, Arc 3 20, Arc 4 40]
   suc 3 = [Arc 4 50]
   suc 2 = [Arc 4 20]
```

Con los métodos definidos podemos calcular

```
caminos 1 4 40 :: [[Int]] ⇝ [[4,1]]
caminos 1 4 50 :: [[Int]] ⇝ [[4,2,1],[4,1]]
caminos 1 4 70 :: [[Int]] ⇝ [[4,2,1],[4,3,1],[4,1]]
```

Notar que se ha indicado el tipo del resultado para resolver las ambiguedades de tipo.

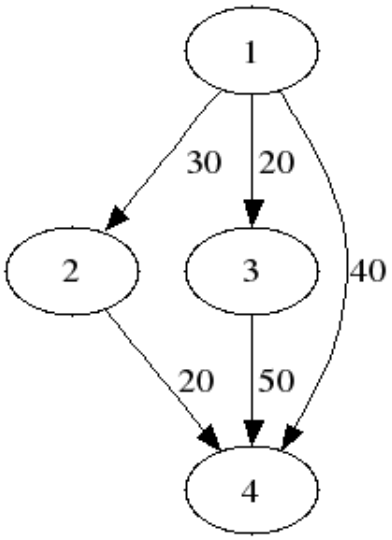


Figura 12.3: Ejemplo de grafo con pesos

12.4.1. El problema de las jarras

Nota. **Enunciado del problema**: En el problema de las jarras `X-Y-Z` se dispone de una jarra de capacidad `X` litros, otra jarra de capacidad `Y` litros, de un grifo que permite llenar las jarras de agua (las jarras no tienen marcas de medición) y de un lavabo donde vaciar las jarras. El problema consiste en averiguar cómo se puede lograr tener `Z` litros de agua en una de las jarras empezando con las dos jarras vacías. Los 8 tipos de acciones que se permiten son llenar una de las jarras con el grifo, llenar una jarra con la otra jarra, vaciar una de las jarras en el lavabo y vaciar una jarra en la otra jarra. La soluciones del problema de las jarras `X-Y-Z` pueden representarse mediante listas de pares `V x y` donde `x` es el contenido de la jarra de `M` litros e `y` el de la de `N` litros. Por ejemplo, una solución del problema `4-3-2` es

```
[J 4 2,J 3 3,J 3 0,J 0 3,J 0 0]
```

es decir, se comienza con las jarras vacías (`J 0 0`), se llena la segunda con el grifo (`J 0 3`), se vacia la segunda en la primera (`J 3 0`), se llena la segunda con el grifo (`J 3 3`) y se llena la primera con la segunda (`J 4 2`).

Nota. Para resolver el problema basta crear una instancia de la clase grafo de búsqueda (`Grafo`) definida en la sección 12.3. Para lo cual basta definir los estados (`Jarras`) y la relación sucesor (`suc`). Una vez creado la instancia, usando el método por defecto `caminosDesde`, se define la función `jarras` que calcula las soluciones.

Nota. El problema se representa en el módulo `Jarras`, que importa el módulo del grafo de búsqueda en profundidad y las funciones de diferencia de conjuntos (`\\`) y eliminación de duplicados (`nub`):

```
module Jarras where
import Grafo_busq_prof
import Data.List ((\\), nub)
```

Nota. Un problema de las jarras es un par (`PJ mx my`) donde `mx` es la capacidad de la primera jarra y `my` la de la segunda.

```
data PJarras = PJ Int Int deriving (Show,Eq)
```

Nota. Un estado en el problema de las jarras es un par (`J x y`) donde `x` es el contenido de la primera jarra e `y` el de la segunda.

```
data Jarras = J Int Int deriving (Show,Eq)
```

Nota. Los operadores de un problema de las jarras son

- `llX`: Llenar la primera jarra con el grifo.
- `llY`: Llenar la segunda jarra con el grifo.
- `vaX`: Vaciar la primera jarra en el lavabo.
- `vaY`: Vaciar la segunda jarra en el lavabo.
- `voXY`: Volcar la primera jarra sobre la segunda, quedando la primera vacía o la segunda llena.
- `voYX`: Volcar la segunda jarra sobre la primera, quedando la segunda vacía o la primera llena.

```
ops :: PJarras -> [Jarras -> Jarras]
ops (PJ mx my) =
    [llX, llY, vaX, vaY, voXY, voYX]
    where llX  (J _ y) = J mx  y
          llY  (J x _) = J x   my
          vaX  (J _ y) = J 0   y
          vaY  (J x _) = J x   0
          voXY (J x y) = if s <= my then J 0 (s) else J (s-my) my
                         where s=x+y
          voYX (J x y) = if s <= mx then J (s) 0 else J mx (s-mx)
                         where s=x+y
```

Nota. Declaración de `Jarras` como instancia de `Grafo`:

```
instance Grafo Jarras where
    suc c = nub [f c | f <- ops (PJ 4 3)] \\ [c]
```

Para cambiar el problema basta modificar la expresión `PJ 4 3`.

Nota. `jarras z` es la lista de soluciones del problema de las jarras para obtener `z` litros en alguna de las jarras. Por ejemplo,

```
Jarras> jarras 2
[[J 4 2,J 3 3,J 3 0,J 0 3,J 4 3,J 4 0,J 0 0],
 [J 4 2,J 3 3,J 3 0,J 0 3,J 4 3,J 1 3,J 4 0,J 0 0],
 [J 4 2,J 3 3,J 3 0,J 0 3,J 1 3,J 4 0,J 0 0],
 [J 4 2,J 3 3,J 3 0,J 0 3,J 4 3,J 4 1,J 0 1,J 1 0,J 1 3,J 4 0,J 0 0],
 [J 2 3,J 4 1,J 0 1,J 1 0,J 1 3,J 4 0,J 0 0],
 [J 4 2,J 3 3,J 3 0,J 0 3,J 0 1,J 1 0,J 1 3,J 4 0,J 0 0],
 [J 2 3,J 4 1,J 0 1,J 1 0,J 1 3,J 4 0,J 4 3,J 0 3,J 0 0],
 [J 2 3,J 4 1,J 0 1,J 1 0,J 1 3,J 4 0,J 3 0,J 0 3,J 0 0],
 [J 2 3,J 4 1,J 0 1,J 1 0,J 1 3,J 4 0,J 4 3,J 3 3,J 3 0,J 0 3,J 0 0],
 [J 4 2,J 3 3,J 3 0,J 0 3,J 0 0]]
```

```
jarras :: Int -> [[Jarras]]
jarras z =
    caminosDesde (J 0 0) test []
    where test (J x y) = x==z || y==z
```

Nota. `másCorta l` es la lista más corta de la lista de listas `l`. Por ejemplo,

```
másCorta [[1,3],[5],[2,3,4]] ⇝ [5]
másCorta (jarras 2)          ⇝ [J 4 2,J 3 3,J 3 0,J 0 3,J 0 0]
```

```
másCorta :: [[a]] -> [a]
másCorta = foldr1 (\ x y -> if length x < length y then x else y)
```

12.4.2. El problema de los misioneros y los caníbales

Nota. **Enunciado del problema**: Hay 3 misioneros y 3 caníbales en la orilla izquierda de un río y tienen una barca en la que caben a lo sumo 2 personas. El problema consiste en diseñar un plan para pasar los 6 a la orilla derecha sin que en ningún momento el número de caníbales que hay en cualquiera de las orillas puede superar al número de misioneros.

Nota. Para resolver el problema basta crear una instancia de la clase grafo de búsqueda (`Grafo`) definida en la sección 12.3. Para lo cual basta definir los estados (`EMisioneros`) y la relación sucesor (`suc`). Una vez creado la instancia, usando el método por defecto `caminosDesde`, se define la función `misioneros` que calcula las soluciones.

Nota. El problema se representa en el módulo `Misioneros`, que importa el módulo del grafo de búsqueda en profundidad y la función `másCorta` del módulo `Jarras`:

```
module Misioneros where
import Grafo_busq_prof
import Jarras (másCorta)
```

Nota. El número de caníbales y de misioneros son números enteros:

```
type Caníbales     = Int
type Misioneros    = Int
```

Nota. La barca puede estar en la derecha o en la izquierda:

```
data PosiciónBarca = Der | Izq deriving (Show,Eq)
```

Nota. Los estados del problema de los misioneros son triples de la forma `EM m c b` donde `m` es el número de misioneros en la orilla izquierda, `c` es el número de caníbales en la orilla izquierda y `b` es la posición de la barca.

```
data EMisioneros = EM Caníbales Misioneros PosiciónBarca
                  deriving (Show,Eq)
```

Nota. `viajes` es la lista de las distintas formas de viajar los misioneros y caníbales suponiendo que la barca no puede viajar vacía ni llevar a más de dos personas.

```
viajes :: [(Misioneros,Caníbales)]
viajes = [(x,y) | x <- [0..2], y <- [0..2], let t=x+y, 0<t, t<=2]
```

Para el presente caso

```
viajes ⇝ [(0,1),(0,2),(1,0),(1,1),(2,0)]
```

Para aumentar la capacidad de la barca basta sustituir 2 por la nueva capacidad.

Nota. `esSegura c m` se verifica si el estando `c` caníbales y `m` misioneros en una orilla, los caníbales no se comen a los misioneros.

```
esSegura :: Caníbales -> Misioneros -> Bool
esSegura _ 0 = True
esSegura c m = m >= c
```

Nota. Declaración de `EMisioneros` como instancia de `Grafo`:

```
instance Grafo EMisioneros where
   suc (EM m c Izq) = [EM m' c' Der |
                       (mb,cb)<- viajes,
                       let m'=m-mb, m'>=0,
                       let c'=c-cb, c'>=0,
                       esSegura c' m',
                       esSegura (3-c') (3-m') ]
   suc (EM m c Der) = [EM m' c' Izq |
                       (mb,cb)<- viajes,
                       let m'=m+mb, m'<=3,
                       let c'=c+cb, c'<=3,
                       esSegura c' m',
                       esSegura (3-c') (3-m') ]
```

Nota. `misioneros` es la lista de soluciones del problema de los misioneros. Por ejemplo,

```
Misioneros> misioneros
[[EM 0 0 Der,EM 0 2 Izq,EM 0 1 Der,EM 0 3 Izq,EM 0 2 Der,EM 2 2 Izq,
  EM 1 1 Der,EM 3 1 Izq,EM 3 0 Der,EM 3 2 Izq,EM 3 1 Der,EM 3 3 Izq],
 [EM 0 0 Der,EM 1 1 Izq,EM 0 1 Der,EM 0 3 Izq,EM 0 2 Der,EM 2 2 Izq,
  EM 1 1 Der,EM 3 1 Izq,EM 3 0 Der,EM 3 2 Izq,EM 3 1 Der,EM 3 3 Izq],
 [EM 0 0 Der,EM 0 2 Izq,EM 0 1 Der,EM 0 3 Izq,EM 0 2 Der,EM 2 2 Izq,
  EM 1 1 Der,EM 3 1 Izq,EM 3 0 Der,EM 3 2 Izq,EM 2 2 Der,EM 3 3 Izq],
 [EM 0 0 Der,EM 1 1 Izq,EM 0 1 Der,EM 0 3 Izq,EM 0 2 Der,EM 2 2 Izq,
  EM 1 1 Der,EM 3 1 Izq,EM 3 0 Der,EM 3 2 Izq,EM 2 2 Der,EM 3 3 Izq]]
```

```
misioneros :: [[EMisioneros]]
misioneros = caminosDesde (EM 3 3 Izq) ((==) (EM 0 0 Der)) []
```

Nota. `misionerosMásCorta` es la solución más corta del problema de los misioneros. Por ejemplo,

```
Misioneros> misionerosMásCorta
[EM 0 0 Der,EM 1 1 Izq,EM 0 1 Der,EM 0 3 Izq,EM 0 2 Der,EM 2 2 Izq,
 EM 1 1 Der,EM 3 1 Izq,EM 3 0 Der,EM 3 2 Izq,EM 2 2 Der,EM 3 3 Izq]
```

```
misionerosMásCorta :: [EMisioneros]
misionerosMásCorta = másCorta misioneros
```

Nota. Todas las soluciones son de la misma longitud. En efecto,

```
map length misioneros ⇝ [12,12,12,12]
```

12.4.3. El problema de reinas

Nota. **Enunciado del problema**: 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.

Nota. Para resolver el problema basta crear una instancia de la clase grafo de búsqueda (`Grafo`) definida en la sección 12.3. Para lo cual basta definir los estados (`EReinas`), la relación sucesor (`suc`) y redefinir el test de visitabilidad (`tv`) ya que en este problema no ocurren ciclos. Una vez creado la instancia, usando el método por defecto `caminosDesde`, se define la función `reinas` que calcula las soluciones.

Nota. El problema se representa en el módulo `Reinas`, que importa el módulo del grafo de búsqueda en profundidad y la diferencia de conjuntos `(\\)` del módulo `List`:

```
module Reinas where
import Grafo_busq_prof
import Data.List ((\\))
```

Nota. Los estados del problema de la reina son pares de la forma `cs :-: ls` donde `cs` es la lista de los números de las filas de las reinas colocadas e `ls` es el número de las filas libres. En la presentación de los estados sólo se escribe la lista de las filas colocadas. Por ejemplo, `[4,2] :-: [1,3]` representa el estado en el que se han colocados las reinas `(1,2)` y `(2,4)` quedando libres las filas 1 y 3.

```
data EReinas = [Int] :-: [Int]  deriving Eq

instance Show EReinas where
    show (x :-: y) = show x
```

Nota. Declaración de `EReinas` como instancia de `Grafo`:

```
instance Grafo EReinas where
   suc ( cs :-: ls) =
       [(f:cs) :-: (ls\\[f]) | f <- ls, no_ataca f cs]
       where no_ataca f cs = and [abs(j-f) /= n | (n,j) <- zip [1..] cs]
   tv _ _ = True --no hay ciclos
```

Nota. `reinas n` es la lista de soluciones del problema de las N reinas. Por ejemplo,

```
reinas 4           ⇝ [[[3,1,4,2],[1,4,2],[4,2],[2],[]],
                                [[2,4,1,3],[4,1,3],[1,3],[3],[]]]
length (reinas 4)  ⇝ 2
map head (reinas ) ⇝ [[3,1,4,2],[2,4,1,3]]
length (reinas 8)  ⇝ 92
```

```
reinas n =
    caminosDesde ([] :-: [1..n]) test []
    where test (_ :-:[]) = True
          test _         = False
```

Nota. Una solución mediante redes de proceso se encuentra en **??**.

Capítulo 13

Juegos

Contenido

13.1. Tres en raya

El tres en raya es un juego entre dos jugadores que marcan los espacios de un tablero de 3x3 alternadamente. Un jugador gana si consigue tener una línea de tres de sus símbolos: la línea puede ser horizontal, vertical o diagonal.
El objetivo de esta sección es realizar un programa para que la máquina juege contra el humano el tres en raya usando la estrategia minimax. Un ejemplo de juego es

```
*Main> main
Tres en raya
1|2|3
-+-+-
4|5|6
-+-+-
7|8|9
Comienza el juego? (s/n) s

Indica el lugar donde colocar la ficha: 5
1|2|3
-+-+-
4|X|6
-+-+-
7|8|9
```

```
Mi jugada:
O|2|3
-+-+-
4|X|6
-+-+-
7|8|9

Indica el lugar donde colocar la ficha: 8
O|2|3
-+-+-
4|X|6
-+-+-
7|X|9

Mi jugada:
O|O|3
-+-+-
4|X|6
-+-+-
7|X|9

Indica el lugar donde colocar la ficha: 3
O|O|X
-+-+-
4|X|6
-+-+-
7|X|9

Mi jugada:
O|O|X
-+-+-
4|X|6
-+-+-
O|X|9

Indica el lugar donde colocar la ficha: 4
O|O|X
-+-+-
X|X|6
```

```
-+-+-
O|X|9

Mi jugada:
O|O|X
-+-+-
X|X|O
-+-+-
O|X|9

Indica el lugar donde colocar la ficha: 9
O|O|X
-+-+-
X|X|O
-+-+-
O|X|X
Empate.
```

Nota. En esta sección se usan las librerías `List` e `IO`.

```
import Data.List
import System.IO
```

Implementación del juego

Ejercicio 13.1.1. *Definir la constante*

```
profundidadDeBusqueda :: Int
```

tal que `profundidadDeBusqueda` *es el máximo nivel de profundidad del árbol de análisis del juego. Por defecto es 6, Cuanto mayor sea la* `profundidadDeBusqueda`*, mejor juega el computador pero su velocidad es menor.*

Solución:

```
profundidadDeBusqueda :: Int
profundidadDeBusqueda = 6
```

Nota. Las posiciones del tablero se numeran como se indica a continuación:

```
1|2|3
-+-+-
4|5|6
-+-+-
7|8|9
```

Ejercicio 13.1.2. *Definir el tipo de dato* `Posicion` *para representar una posición del tablero. Cada posición es un entero del 1 a 9.*

Solución:

```
type Posicion = Int
```

Ejercicio 13.1.3. *Definir el tipo de datos* `Posiciones` *para representar listas de posiciones.*

Solución:

```
type Posiciones = [Posicion]
```

Nota. Hay dos jugadores. El jugador X es el que comienza el juego y el otro jugador es el O.

Ejercicio 13.1.4. *Definir el tipo de datos* `Tablero` *para representar los tableros. El tablero de la forma* `(Tab xs os)` *representa un tablero donde* `xs` *es la lista de las posiciones donde están colocadas las fichas del primer jugador y* `os` *la del segundo jugador.*

Solución:

```
data Tablero = Tab Posiciones Posiciones
               deriving Show
```

Ejercicio 13.1.5. *Definir la constante*

```
tableroInicial :: Tablero
```

par representar el tablero inicial.

Solución:

```
tableroInicial :: Tablero
tableroInicial = Tab [] []
```

Ejercicio 13.1.6. *Definir la función*

```
turnoDeX :: Tablero -> Bool
```

tal que `(turnoDeX t)` *se verifica si en el tablero* `t` *le toca mover al jugador* `X`.

Solución:

```
turnoDeX :: Tablero -> Bool
turnoDeX (Tab xs os) =
    (length xs == length os)
```

Ejercicio 13.1.7. *Definir la función*

```
pone :: Tablero -> Posicion -> Tablero
```

tal que `(pone t p)` *es el tablero obtenido poniendo en la posición* `p` *del tablero* `t` *una ficha del jugador al que le correponde colocar.*

Solución:

```
pone :: Tablero -> Posicion -> Tablero
pone (Tab xs os) p =
    if turnoDeX (Tab xs os)
    then (Tab (p:xs) os)
    else (Tab xs (p:os))
```

Ejercicio 13.1.8. *Definir la función*

```
completo :: Tablero -> Bool
```

tal que `(completo t)` *se verifica si el tablero* `t` *está completo; es decir, se han colocado las 9 fichas.*

Solución:

```
completo :: Tablero -> Bool
completo (Tab xs os) =
    length xs + length os == 9
```

Ejercicio 13.1.9. *Definir la función*

```
subconjunto :: Posiciones -> Posiciones -> Bool
```

tal que `(subconjunto s1 s2)` *se verifica si* `s1` *es un subconjunto de* `s2`*.*

Solución:

```
subconjunto :: Posiciones -> Posiciones -> Bool
subconjunto s1 s2 =
    all (`elem` s2) s1
```

Ejercicio 13.1.10. *Definir la función*

```
tieneLinea :: Posiciones -> Bool
```

tal que `(tieneLinea ps)` *se verifica si la lista de posiciones* `ps` *contiene una línea horizontal, vertical o diagonal.*

Solución:

```
tieneLinea :: Posiciones -> Bool
tieneLinea ps =
    subconjunto [1,2,3] ps ||subconjunto [4,5,6] ps ||subconjunto [7,8,9] ps ||
    subconjunto [1,4,7] ps ||subconjunto [2,5,8] ps ||subconjunto [3,6,9] ps ||
    subconjunto [1,5,9] ps ||subconjunto [3,5,7] ps
```

Ejercicio 13.1.11. *Definir la función*

```
tieneGanador :: Tablero -> Bool
```

tal que `(tieneGanador t)` *se verifica si el tablero* `t` *tiene un ganador; es decir, alguno de los dos jugadores ha conseguido una línea.*

Solución:

```
tieneGanador :: Tablero -> Bool
tieneGanador (Tab xs os) =
    tieneLinea xs || tieneLinea os
```

Construcción del árbol de juego

Ejercicio 13.1.12. *Definir el tipo de datos* `Arbol` *para representa los árboles compuestos por nodos con una lista de hijos.*

Solución:

```
data Arbol a = Nodo a [Arbol a]
```

Ejercicio 13.1.13. *Definir la función*

```
muestraArbol :: Show t => Arbol t -> String
```

tal que `(muestraArbol t)` *es una cadena que representa el árbol* `t` *para una mejor visualización. Hacer la clase* `Arbol` *una istancia de* `Show` *definiendo* `show` *como* `muestraArbol`*. Por ejemplo,*

```
Main> muestraArbol (Nodo 1 [Nodo 2 [Nodo 4 []], Nodo 3 []])
"1\n  2\n    4\n  3\n"
Main> Nodo 1 [Nodo 2 [Nodo 4 []], Nodo 3 []]
1
  2
    4
  3
```

Solución:

```
muestraArbol (Nodo x xs) =
    show x ++ '\n' :  (unlines . map ("  "++) . concatMap (lines . show)) xs

instance Show a => Show (Arbol a) where
  show = muestraArbol
```

En la siguiente sesión se muestra el comportamiento de `muestraArbol`.

```
Main> show 1
"1"
Main> concatMap (lines . show) [Nodo 2 [Nodo 4 []], Nodo 3 []]
["2","  4","3"]
Main> map ("  "++) ["2","  4","3"]
["  2","    4","  3"]
Main> unlines ["  2","    4","  3"]
"  2\n    4\n  3\n"
Main> "1" ++ '\n' : "  2\n    4\n  3\n"
"1\n  2\n    4\n  3\n"
```

Ejercicio 13.1.14. *Definir la función*

```
posicionesLibres :: Tablero -> Posiciones
```

tal que `(posicionesLibres t)` *es la lista de las posiciones libres del tablero* `t`.

Solución:

```
posicionesLibres :: Tablero -> Posiciones
posicionesLibres (Tab xs os) =
    [1..9] \\ (xs++os)
```

Ejercicio 13.1.15. *Definir la función*

```
siguientesTableros :: Tablero -> [Tablero]
```

tal que `(siguientesTableros t)` *es la lista de tableros obtenidos colocando una pieza en cada una de las posiciones libres de* `t`. *Por ejemplo,*

```
Main> tableroInicial
Tab [] []
Main> siguientesTableros tableroInicial
[Tab [1] [], Tab [2] [], Tab [3] [], Tab [4] [], Tab [5] [],
 Tab [6] [], Tab [7] [], Tab [8] [], Tab [9] []]
```

```
Main> siguientesTableros (Tab [1] [])
[Tab [1] [2], Tab [1] [3], Tab [1] [4], Tab [1] [5],
 Tab [1] [6], Tab [1] [7], Tab [1] [8], Tab [1] [9]]
Main> siguientesTableros (Tab [1] [2])
[Tab [3,1] [2], Tab [4,1] [2], Tab [5,1] [2], Tab [6,1] [2],
 Tab [7,1] [2], Tab [8,1] [2], Tab [9,1] [2]]
```

Solución:

```
siguientesTableros :: Tablero -> [Tablero]
siguientesTableros t =
    if tieneGanador t
    then []
    else map (pone t) (posicionesLibres t)
```

Ejercicio 13.1.16. *Definir la función*

```
construyeArbol :: Tablero -> Arbol Tablero
```

tal que `(construyeArbol t)` *es el árbol de juego correspondiente al tablero* `t`*. Por ejemplo,*

```
Main> construyeArbol (Tab [7,1,6,2] [5,4,3])
 Tab [7,1,6,2] [5,4,3]
   Tab [7,1,6,2] [8,5,4,3]
     Tab [9,7,1,6,2] [8,5,4,3]
   Tab [7,1,6,2] [9,5,4,3]
     Tab [8,7,1,6,2] [9,5,4,3]
```

Solución:

```
construyeArbol :: Tablero -> Arbol Tablero
construyeArbol t =
    Nodo t (map construyeArbol (siguientesTableros t))
```

Ejercicio 13.1.17. *Definir el tipo* `Valor` *para representa el valor de los tableros. Los valores son números enteros.*

Solución:

```
type Valor = Int
```

Nota. Un tablero valorado es un par de la forma `(v,t)` donde `t` es un tablero y `v` es el valor del tablero.

Ejercicio 13.1.18. *Definir la función*

```
valores :: [Arbol (Valor,Tablero)] -> [Valor]
```

tal que `(valores vts)` *es la lista de valores de la lista de árboles de tableros valorados* `vts`*.*

Solución:

```
valores :: [Arbol (Valor,Tablero)] -> [Valor]
valores vts =
    [v | Nodo (v,_) _ <- vts]
```

Ejercicio 13.1.19. *Definir la función*

```
maximiza :: Arbol Tablero -> Arbol (Valor,Tablero)
```

tal que `(maximiza at)` *es el árbol de tableros máximamente valorados correspondiente al árbol de tableros* `at`*.*

Solución:

```
maximiza :: Arbol Tablero -> Arbol (Valor,Tablero)
maximiza (Nodo t []) = Nodo (if tieneGanador t then -1 else 0,t) []
maximiza (Nodo t ts) = Nodo (maximum (valores vts),t) vts
    where vts = map minimiza ts
```

Ejercicio 13.1.20. *Definir la función*

```
minimiza :: Arbol Tablero -> Arbol (Valor,Tablero)
```

tal que `(minimiza at)` *es el árbol de tableros mínimamente valorados – correspondiente al árbol de tableros* `at`*.*

Solución:

```
minimiza :: Arbol Tablero -> Arbol (Valor,Tablero)
minimiza (Nodo t []) = Nodo (if tieneGanador t then 1 else 0,t) []
minimiza (Nodo t ts) = Nodo (minimum (valores vts),t) vts
    where vts = map maximiza ts
```

Ejercicio 13.1.21. *Definir la función*

```
poda :: Int -> Arbol a -> Arbol a
```

tal que `(poda n a)` *es el árbol obtenido podando el árbol* `a` *a partir de la profundidad* `n`*.*

Solución:

```
poda :: Int -> Arbol a -> Arbol a
poda n (Nodo x xs) =
    Nodo x (if n==0
            then []
            else (map (poda (n-1)) xs))
```

Ejercicio 13.1.22. *Definir la función*

```
selecciona :: Arbol (Valor,Tablero) -> Tablero
```

tal que `(selecciona avts)` *es el tablero del primer hijo de la raíz del árbol de tableros valorados* `avts` *cuyo valor es igual que la raiz.*

Solución:

```
selecciona :: Arbol (Valor,Tablero) -> Tablero
selecciona (Nodo (v,_) ts) =
    head [t | Nodo (v',t) _ <- ts, v'==v]
```

Ejercicio 13.1.23. *Definir la función*

```
mejorMovimiento :: Tablero -> Tablero
```

tal que `(mejorMovimiento t)` *es el tablero correspondiente al mejor movimiento a partir del tablero* `t`.

Solución:

```
mejorMovimiento :: Tablero -> Tablero
mejorMovimiento =
    selecciona . maximiza . poda profundidadDeBusqueda . construyeArbol
```

Dibujo del tablero

Ejercicio 13.1.24. *Definir la función*

```
muestraPosicion :: Tablero -> Posicion -> String
```

tal que `(muestraPosicion t p)` *es el contenido de la posición* `p` *del tablero* `t`*; es decir,* `X` *si* `p` *está en la lista de las* `xs`*;* `O` *si* `p` *está en la lista de las* `os` *y la cadena de* `p`*, en otro caso. Por ejemplo,*

```
Main> muestraPosicion (Tab [1] [3]) 1
"X"
Main> muestraPosicion (Tab [1] [3]) 3
"O"
Main> muestraPosicion (Tab [1] [3]) 2
"2"
```

Solución:

```
muestraPosicion :: Tablero -> Posicion -> String
muestraPosicion (Tab xs os) p
    | p `elem` xs = "X"
    | p `elem` os = "O"
    | otherwise   = show p
```

Ejercicio 13.1.25. *Definir la función*

```
muestraLinea :: Tablero -> [Posicion] -> String
```

tal que `(muestraLinea t ps)` *es la cadena correspondiente al contenido de las posiciones* `ps` *en el tablero* `t` *separadas por la barra vertical. Por ejemplo,*

```
Main> muestraLinea (Tab [7,1,6,2] [8,4,3]) [4..6]
"O|5|X"
```

Solución:

```
muestraLinea :: Tablero -> [Posicion] -> String
muestraLinea t =
    concat . intersperse "|" . map (muestraPosicion t)
```

Ejercicio 13.1.26. *Definir la función*

```
muestraTablero :: Tablero -> String
```

tal que `(muestraTablero t)` *es la cadena correspondiente al tablero* `t`*. Por ejemplo,*

```
Main> muestraTablero (Tab [7,1,6,2] [8,4,3])
"X|X|O\n-+-+-\nO|5|X\n-+-+-\nX|O|9"
Main> putStrLn (muestraTablero (Tab [7,1,6,2] [8,4,3]))
X|X|O
-+-+-
O|5|X
-+-+-
X|O|9
```

Solución:

```
muestraTablero :: Tablero -> String
muestraTablero t =
    muestraLinea t [1..3] ++ "\n-+-+-\n" ++
    muestraLinea t [4..6] ++ "\n-+-+-\n" ++
    muestraLinea t [7..9]
```

Control del juego

Ejercicio 13.1.27. *Definir la función*

```
main :: IO ()
```

que controle el juego siguiendo los siguientes pasos:

1. *Activa la escritura inmediata en la pantalla.*
2. *Escribe el nombre del juego.*
3. *Escribe el tablero inicial.*
4. *Pregunta al humano si desea comenzar el juego.*
5. *Para y lee la respuesta.*
6. *Comprueba si la respuesta es afirmativa.*
7. *En el caso que la respuesta sea afirmativa, realiza un movimiento del jugador humano.*
8. *En el caso que la respuesta sea negativa, realiza un movimiento de la computadora.*

Solución:

```
main :: IO ()
main = do
  hSetBuffering stdout NoBuffering          -- 1
  putStrLn "Tres en raya"                   -- 2
  putStrLn (muestraTablero tableroInicial)  -- 3
  putStr "Comienza el juego? (s/n) "        -- 4
  l <- getLine                              -- 5
  if head l `elem` ['s','S']                -- 6
     then humano tableroInicial             -- 7
     else computadora tableroInicial        -- 8
```

Ejercicio 13.1.28. *Definir la función*

```
humano :: Tablero -> IO ()
```

tal que `(humano t)` *realiza el movimiento del jugador humano a partir del tablero* `t`*. Consta de los siguientes pasos:*

1. *Pregunta la posición en donde desea colocar la ficha.*
2. *Lee la posición en donde desea colocar la ficha.*
3. *Calcula el tablero* `t'` *correspondiente a colocar la ficha en la posición elegida.*
4. *Muestra el tablero* `t'`*.*
5. *Decide si* `t'` *tiene ganador.*
   a) *En caso afirmativo, escribe que el jugador humano ha ganado.*
   b) *En caso negativo, decide si el tablero está completo*
      1) *En caso afirmativo, escribe que hay empate.*
      2) *En caso negativo, pasa el turno a la computadora con tablero* `t'`*.*

Nota: *No se comprueba la corrección de la posición elegida (es decir, si es un número entre 1 y 9 y no hay ficha en esa posición).*

Solución:

```
humano :: Tablero -> IO ()
humano t = do
  putStr "\nIndica el lugar donde colocar la ficha: " -- 1
  l <- getLine                                        -- 2
  let t' = pone t (read l :: Posicion)                -- 3
  putStrLn (muestraTablero t')                        -- 4
  if tieneGanador t'                                  -- 5
     then putStrLn "Has ganado."                      -- 5.a
     else if (completo t')                            -- 5.b
             then putStrLn "Empate."                  -- 5.b.1
             else computadora t'                      -- 5.b.2
```

Ejercicio 13.1.29. *Definir la función*

```
computadora :: Tablero -> IO ()
```

tal que `(computadora t)` *realiza el movimiento de la computadora a partir del tablero* `t`*. Consta de los siguientes pasos:*

1. *Escribe la jugada de la computadora*
2. *Calcula el tablero* `t'` *correspondiente al mejor movimiento en* `t`.
3. *Escribe* `t'`.
4. *Decide si* `t'` *tiene ganador.*
   a) *En caso afirmativo, escribe que la computadora ha ganado.*
   b) *En caso negativo, decide si el tablero está completo.*
      1) *En caso afirmativo, escribe que hay empate.*
      2) *En caso negativo, pasa el turno al humano con tablero* `t'`.

Solución:

```
computadora :: Tablero -> IO ()
computadora t = do
  putStrLn "\nMi jugada:"            -- 1
  let t' = mejorMovimiento t         -- 2
  putStrLn (muestraTablero t')       -- 3
  if tieneGanador t'                 -- 4
     then putStrLn "He ganado."      -- 4.a
     else if (completo t')           -- 4.b
             then putStrLn "Empate." -- 4.b.1
             else humano t'          -- 4.b.2
```