Prácticas de "Programación Declarativa" (2007–08)

José A. Alonso Jiménez

Grupo de Lógica Computacional
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, 16 de Octubre de 2007 (Versión de 17 de diciembre de 2007)

Índice general

Capítulo 1

Práctica 1a (22 de octubre de 2007)

1.1. Práctica 1a (22 de octubre de 2007)

```
-- ---------------------------------------------------------------------
-- Importación de librerías auxiliares                                 --
-- ---------------------------------------------------------------------

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1 (Transformación entre euros y pesetas)
-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Calcular cuántas pesetas son 49 euros (1 euro son 166.386
-- pesetas).
-- ---------------------------------------------------------------------

-- El cálculo es
--    Hugs> 49*166.386
--    8152.914

-- ---------------------------------------------------------------------
-- Ejercicio 1.2, Definir la constante cambioEuro cuyo valor es 166.386 y
-- repetir el cálculo anterior usando la constante definida.
-- ---------------------------------------------------------------------

-- La definición es
cambioEuro = 166.386

-- y el cálculo es
```

```
--    Main> 49*cambioEuro
--    8152.914

-- ---------------------------------------------------------------------
-- Ejercicio 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.
-- ---------------------------------------------------------------------

-- La definición es
pesetas x = x*cambioEuro

-- y el cálculo es
--    Main> pesetas 49
--    8152.914

-- ---------------------------------------------------------------------
-- Ejercicio 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.
-- ---------------------------------------------------------------------

-- La definición es
euros x = x/cambioEuro

-- y el cálculo es
--    Main> euros 8152.914
--    49.0

-- ---------------------------------------------------------------------
-- Ejercicio 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.
-- ---------------------------------------------------------------------

-- La definición es
prop_EurosPesetas x =
    euros(pesetas x) == x
```

```
-- y la comprobación es
--    Main> prop_EurosPesetas 49
--    True

-- ---------------------------------------------------------------------
-- Ejercicio 1.6. Comprobar la prop_EurosPesetas con QuickCheck.
-- ---------------------------------------------------------------------

-- 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 1.7. Calcular la diferencia entre euros(pesetas 3.625) y 3.625.
-- ---------------------------------------------------------------------

-- El cálculo es
--    Main> euros(pesetas 3.625)-3.625
--    -4.44089209850063e-16

-- ---------------------------------------------------------------------
-- Ejercicio 1.8. Definir el operador ~= tal que (x ~= y) se verifique si el
-- valor absoluto de la diferencia entre x e y es menor que una milésima. Se
-- dice que x e y son casi iguales.
-- ---------------------------------------------------------------------

-- La definición es
x ~= y = abs(x-y)<0.001

-- ---------------------------------------------------------------------
-- Ejercicio 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.
-- ---------------------------------------------------------------------
```

```
-- La definición es
prop_EurosPesetas' x =
    euros(pesetas x) ~= x

-- y la comprobación es
--    Main> prop_EurosPesetas' 49
--    True

-- ---------------------------------------------------------------------
-- Ejercicio 1.10. Comprobar la prop_EurosPesetas' con QuickCheck.
-- ---------------------------------------------------------------------

-- La comprobación es
--    Main> quickCheck prop_EurosPesetas'
--    OK, passed 100 tests.
-- lo que indica que se cumple para los 100 casos de pruebas considerados.

-- ---------------------------------------------------------------------
-- Ejercicio 2 (Máximo de enteros)
-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Definir, por casos, la función maxI 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
-- ---------------------------------------------------------------------

-- La definición es
maxI :: Integer -> Integer -> Integer
maxI x y | x >= y    = x
         | otherwise = y

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. En este apartado vamos a comprobar propiedades del máximo.
-- ---------------------------------------------------------------------
-- Ejercicio 2.2.1. Comprobar con QuickCheck que el máximo de x e y es
-- mayor o igual que x y que y.
-- ---------------------------------------------------------------------

-- La propiedad es
```

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

-- y la comprobación es
--    Main> quickCheck prop_MaxIMayor
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 2.2.2. Comprobar con QuickCheck que el máximo de x e y es x ó y.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_MaxIAlguno x y =
  maxI x y == x || maxI x y == y

-- y la comprobación es
--    Main> quickCheck prop_MaxIAlguno
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 2.2.3. Comprobar con QuickCheck que si x es mayor o igual
-- que y, entonces el máximo de x e y es x.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_MaxIX x y =
  x >= y ==> maxI x y == x

-- y la comprobación es
--    Main> quickCheck prop_MaxIX
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 2.2.4. Comprobar con QuickCheck que si y es mayor o igual
-- que x, entonces el máximo de x e y es y.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_MaxIY x y =
  y >= x ==>  maxI x y == y
```

```
-- y la comprobación es
--    Main> quickCheck prop_MaxIY
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 3 (Suma de cuadrados)
-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir por recursión la función sumaCuadrados tal que
-- (sumaCuadrados n) es la suma de los cuadrados de los números de 1 a n. Por
-- ejemplo,
--    sumaCuadrados 4 => 30
-- ---------------------------------------------------------------------

-- La definición es
sumaCuadrados :: Integer -> Integer
sumaCuadrados 0         = 0
sumaCuadrados n | n > 0 = sumaCuadrados (n-1) + n*n

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Comprobar con QuickCheck si sumaCuadrados n es igual a
-- n(n+1)(2n+1)/6.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_SumaCuadrados n =
  n >= 0 ==>
    sumaCuadrados n == n * (n+1) * (2*n+1) `div` 6

-- y la comprobación es
--    Main> quickCheck prop_SumaCuadrados
--    OK, passed 100 tests.
```

Capítulo 2

Práctica 1b (29 de octubre de 2007)

2.1. Práctica 1b (29 de octubre de 2007)

```
-- ---------------------------------------------------------------------------
-- Importación de librerías auxiliares                                       --
-- ---------------------------------------------------------------------------

import Test.QuickCheck

-- ---------------------------------------------------------------------------
-- Ejercicio 1 (Transformación entre euros y pesetas)
-- ---------------------------------------------------------------------------
-- Ejercicio 1.1. Calcular cuántas pesetas son 49 euros (1 euro son 166.386
-- pesetas).
-- ---------------------------------------------------------------------------

-- El cálculo es
--    Hugs> 49*166.386
--    8152.914

-- ---------------------------------------------------------------------------
-- Ejercicio 1.2, Definir la constante cambioEuro cuyo valor es 166.386 y
-- repetir el cálculo anterior usando la constante definida.
-- ---------------------------------------------------------------------------

-- La definición es
cambioEuro = 166.386

-- y el cálculo es
```

```
--    Main> 49*cambioEuro
--    8152.914

-- ---------------------------------------------------------------------
-- Ejercicio 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.
-- ---------------------------------------------------------------------

-- La definición es
pesetas x = x*cambioEuro

-- y el cálculo es
--    Main> pesetas 49
--    8152.914

-- ---------------------------------------------------------------------
-- Ejercicio 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.
-- ---------------------------------------------------------------------

-- La definición es
euros x = x/cambioEuro

-- y el cálculo es
--    Main> euros 8152.914
--    49.0

-- ---------------------------------------------------------------------
-- Ejercicio 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.
-- ---------------------------------------------------------------------

-- La definición es
prop_EurosPesetas x =
    euros(pesetas x) == x
```

```
-- y la comprobación es
--    Main> prop_EurosPesetas 49
--    True

-- ---------------------------------------------------------------------
-- Ejercicio 1.6. Comprobar la prop_EurosPesetas con QuickCheck.
-- ---------------------------------------------------------------------

-- 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 1.7. Calcular la diferencia entre euros(pesetas 3.625) y 3.625.
-- ---------------------------------------------------------------------

-- El cálculo es
--    Main> euros(pesetas 3.625)-3.625
--    -4.44089209850063e-16

-- ---------------------------------------------------------------------
-- Ejercicio 1.8. Definir el operador ~= tal que (x ~= y) se verifique si el
-- valor absoluto de la diferencia entre x e y es menor que una milésima. Se
-- dice que x e y son casi iguales.
-- ---------------------------------------------------------------------

-- La definición es
x ~= y = abs(x-y)<0.001

-- ---------------------------------------------------------------------
-- Ejercicio 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.
-- ---------------------------------------------------------------------
```

```
-- La definición es
prop_EurosPesetas' x =
    euros(pesetas x) ~= x

-- y la comprobación es
--    Main> prop_EurosPesetas' 49
--    True

-- ---------------------------------------------------------------------
-- Ejercicio 1.10. Comprobar la prop_EurosPesetas' con QuickCheck.
-- ---------------------------------------------------------------------

-- La comprobación es
--    Main> quickCheck prop_EurosPesetas'
--    OK, passed 100 tests.
-- lo que indica que se cumple para los 100 casos de pruebas considerados.

-- ---------------------------------------------------------------------
-- Ejercicio 2 (Menor múltiplo mayor o igual que)
-- ---------------------------------------------------------------------
-- El objetivo de este ejercicio consiste en definir la función
-- menorMúltiplo tal que (menorMúltiplo n p) es el menor número mayor o
-- igual que n que es múltiplo de p. Por ejemplo,
--    menorMúltiplo 16 5   ==> 20
--    menorMúltiplo 167 25 ==> 175
-- Para ello, partiremos de la siguiente propiedad
--    menorMúltiplo n p = n + (p - (n resto p))
-- es decir, n más la diferencia entre p y el resto de la división entre
-- n y p.
-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Definir la función menorMúltiplo.
-- ---------------------------------------------------------------------

-- La definición es
menorMúltiplo :: Integer -> Integer -> Integer
menorMúltiplo n p =  n + (p - (n `rem` p))

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Calcular el resultado de menorMúltiplo para los ejemplos
```

```
-- anteriores.
-- ---------------------------------------------------------------------

-- Los cálculos son:
--    menorMúltiplo 16 5    ==>  20
--    menorMúltiplo 167 25  ==>  175

-- ---------------------------------------------------------------------
-- Ejercicio 2.3. En este apartado vamos a estudiar propiedades de la función
-- menorMúltiplo.
-- ---------------------------------------------------------------------
-- Ejercicio 2.3.1. Comprobar con QuickCheck si (menorMúltiplo n p) es
-- mayor o igual que n.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_menorMúltiplo_1 n p =
    menorMúltiplo n p >= n

-- y la comprobación es
--    Main> quickCheck prop_menorMúltiplo_1
--    Falsifiable, after 0 tests:
--    -2
--    -3

-- La propiedad no se verifica. Pero sí se verifica cuando los parámetros son
-- positivos:
prop_menorMúltiplo_1' n p =
    n>0 && p>0 ==> menorMúltiplo n p >= n

-- En efecto,
--    Main> quickCheck prop_menorMúltiplo_1'
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 2.3.2. Comprobar con QuickCheck si (menorMúltiplo n p) es
-- múltiplo de p.
-- ---------------------------------------------------------------------

-- La propiedad es
```

```
prop_menorMúltiplo_2 n p =
    n>0 && p>0 ==> múltiplo (menorMúltiplo n p) p

-- donde (múltiplo n p) se verifica si n es un múltiplo de p.
múltiplo :: Integer -> Integer -> Bool
múltiplo n p = n `mod` p == 0

-- La comprobación es
--    Main> quickCheck prop_menorMúltiplo_2
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 2.3.3. Comprobar con QuickCheck si (menorMúltiplo n n) es
-- igual a n.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_menorMúltiplo_3 n =
    n>0 ==> menorMúltiplo n n == n

-- La comprobación es
--    Main> quickCheck prop_menorMúltiplo_3
--    Falsifiable, after 0 tests:
--    1

-- La propiedad no se verifica. En efecto,
--    menorMúltiplo 1 1  ==>  2

-- ---------------------------------------------------------------------
-- Ejercicio 2.4. Corregir los errores de la definición de menorMúltiplo
-- escribiendo una nueva función menorMúltiplo'. Definir las correspondientes
-- propiedades y comprobarlas.
-- ---------------------------------------------------------------------

-- La definición es
menorMúltiplo' :: Integer -> Integer -> Integer
menorMúltiplo' n p | múltiplo n p = n
                   | otherwise    = n + (p - (n `rem` p))

-- Las propiedades son
```

```
prop_menorMúltiplo'_1 n p =
    n>0 && p>0 ==> menorMúltiplo' n p >= n

prop_menorMúltiplo'_2 n p =
    n>0 && p>0 ==> múltiplo (menorMúltiplo' n p) p

prop_menorMúltiplo'_3 n =
    n>0 ==> menorMúltiplo' n n == n

-- y sus comprobaciones son
--    Main> quickCheck prop_menorMúltiplo'_1
--    OK, passed 100 tests.
--
--    Main> quickCheck prop_menorMúltiplo'_2
--    OK, passed 100 tests.
--
--    Main> quickCheck prop_menorMúltiplo'_3
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 2.5. Obtener una definición recursiva que se denomine
-- menorMúltiplo'', definir las correspondientes propiedades y comprobarlas.
-- ---------------------------------------------------------------------

-- La definición es
menorMúltiplo'' :: Integer -> Integer -> Integer
menorMúltiplo'' n p | múltiplo n p = n
                    | otherwise    = menorMúltiplo'' (n+1) p

-- Las propiedades son
prop_menorMúltiplo''_1 n p =
    n>0 && p>0 ==> menorMúltiplo'' n p >= n

prop_menorMúltiplo''_2 n p =
    n>0 && p>0 ==> múltiplo (menorMúltiplo'' n p) p

prop_menorMúltiplo''_3 n =
    n>0 ==> menorMúltiplo'' n n == n

-- y las comprobaciones son
```

```
--    Main> quickCheck prop_menorMúltiplo''_1
--    OK, passed 100 tests.
--
--    Main> quickCheck prop_menorMúltiplo''_2
--    OK, passed 100 tests.
--
--    Main> quickCheck prop_menorMúltiplo''_3
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 3 (Máximo común divisor)
-- ---------------------------------------------------------------------
-- Dados dos números naturales, a y b, es posible calcular su máximo
-- común divisor mediante el Algoritmo de Euclides. Este algoritmo se
-- puede resumir en la siguiente fórmula:
--    mcd(a,b) = a,                    si b = 0
--             = mcd (b, a módulo b), si b > 0
-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir la función mcd.
-- ---------------------------------------------------------------------

-- La definición es
mcd :: Integer -> Integer -> Integer
mcd a 0 = a
mcd a b = mcd b (a `mod` b)

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir y comprobar la propiedad prop_mcd según la cual el
-- máximo común divisor de dos números a y b (ambos mayores que 0) es siempre
-- mayor o igual que 1 y además es menor o igual que el menor de los números a
-- y b.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_mcd a b =
    a>0 && b>0 ==> m>=1 && m <= min a b
                   where m = mcd a b

-- y su comprobación es
--    Main> quickCheck prop_mcd
```

```
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------------
-- Ejercicio 3.3. Teniendo en cuenta que buscamos el máximo común divisor de a
-- y b, sería razonable pensar que el máximo común divisor siempre sería igual
-- o menor que la mitad del máximo de a y b. Definir esta propiedad y
-- comprobarla.
-- ---------------------------------------------------------------------------

-- La propiedad es
prop_mcd_div a b =
    a > 0 && b > 0 ==> mcd a b <= (max a b) `div` 2

-- Al verificarla, se obtiene
--    Main> quickCheck prop_mcd_div
--    Falsifiable, after 0 tests:
--    3
--    3
-- que la refuta. Pero si la modificamos añadiendo la hipótesis que los números
-- son distintos,
prop_mcd_div' a b =
    a > 0 && b > 0 && a /= b ==> mcd a b <= (max a b) `div` 2

-- entonces al comprobarla
--    Main> quickCheck prop_mcd_div'
--    OK, passed 100 tests.
-- obtenemos que se verifica.
```

Capítulo 3

Práctica 2a (5 de noviembre de 2007)

3.1. Práctica 2a (5 de noviembre de 2007)

```
-- ---------------------------------------------------------------------
-- Importación de librerías auxiliares                                 --
-- ---------------------------------------------------------------------

import Data.List
import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1. (Medidas de centralización)
-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir la función
--    media :: [Double] -> Double
-- que calcule la media aritmética de una lista numérica. Por ejemplo,
--    media [1,2,3]       ==>  2.0
--    media [1,-2,3.5,4]  ==>  1.625
-- Nota: Usar la predefinida fromIntegral.
-- ---------------------------------------------------------------------

media :: [Double] -> Double
media xs = (sum xs) / fromIntegral (length xs)

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir la función
--    mediana :: [Double] -> Double
-- que calcule la mediana de una lista numérica; esto es, el valor que deja el
-- mismo número de datos menores y mayores que él (si la lista contiene un
```

```
-- número par de elementos, la mediana será la media aritmética de los dos
-- valores centrales). Por ejemplo,
--    mediana [3,2,5,1,4]  ==>  3.0
--    mediana [-1,7,8,1]   ==>  4.0
-- Nota: Usar la función sort de la biblioteca Data.List
-- ---------------------------------------------------------------------

mediana :: [Double] -> Double
mediana xs | odd a     = ys !! ((a-1) `div` 2)
           | otherwise = (ys !! (a `div`2) + ys !! (a `div` 2 - 1) ) / 2
    where a  = length xs
          ys = sort xs

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. La función
--    divideMedia :: [Double] -> [[Double]]
-- dada una lista numérica, xs, calcula la lista [ys,zs], donde ys contiene los
-- elementos de xs estrictamente menores que la media, mientras que zs contiene
-- los elementos de xs estrictamente mayores que la media. Por ejemplo,
--    divideMedia [6,7,2,8,6,3,4]  ==>  [[2.0,3.0,4.0],[6.0,7.0,8.0,6.0]]
--    divideMedia [1,2,3]          ==>  [[1.0],[3.0]]
-- En los distintos subapartados se piden distintas definiciones de la función
-- divideMedia.
-- ---------------------------------------------------------------------
-- Ejercicio 1.3.1. Definir la función divideMedia por recursión sobre listas.
-- ---------------------------------------------------------------------

divideMedia1 :: [Double] -> [[Double]]
divideMedia1 xs = divideMedia1Aux (media xs) xs

divideMedia1Aux :: Double -> [Double] -> [[Double]]
divideMedia1Aux _ [] = [[],[]]
divideMedia1Aux c (x:xs)
    | x < c     = [(x:ys),zs]
    | x > c     = [ys,(x:zs)]
    | otherwise = [ys,zs]
    where [ys,zs] = divideMedia1Aux c xs

-- ---------------------------------------------------------------------
-- Ejercicio 1.3.2. Definir la función divideMedia por filtrado.
```

```
-- ---------------------------------------------------------------------

divideMedia2 :: [Double] -> [[Double]]
divideMedia2 xs = [filter (<m) xs, filter (>m) xs]
    where m = media xs

-- ---------------------------------------------------------------------
-- Ejercicio 1.3.3. Definir la función divideMedia por comprensión.
-- ---------------------------------------------------------------------

divideMedia3 :: [Double] -> [[Double]]
divideMedia3 xs = [[x | x <- xs, x < m],[x | x <- xs, x > m]]
    where m = media xs

-- ---------------------------------------------------------------------
-- Ejercicio 1.4. Dada una lista, xs, sus valores externos son aquellos valores
-- menores que media xs-(máximo xs-mínimo xs)/2) o mayores que
-- media xs+(máximo xs-mínimo xs)/2).
--
-- En los distintos subapartados se piden distintas definiciones de la función
--    valoresExternos:: [Double] -> [Double]
-- que calcula los valores externos de una lista numérica. Por ejemplo,
--   valoresExternos [1,2,5,5,5,5,5,5]      ==>  [1.0,2.0]
--   valoresExternos [5,5,5,5,5,5,8,9]      ==>  [8.0,9.0]
--   valoresExternos [1,2,5,5,5,5,5,5,8,9]  ==>  []
-- ---------------------------------------------------------------------
-- Ejercicio 1.4.1. Definir la función valoresExternos por recursión.
-- ---------------------------------------------------------------------

valoresExternos1 :: [Double] -> [Double]
valoresExternos1 xs = aux1 (media xs) (minimum xs) (maximum xs) xs
    where aux1 _ _ _[]                     = []
          aux1 m a b (x:xs)
              | x<m-(b-a)/2 || x>m+(b-a)/2 = x:(aux1 m a b xs)
              | otherwise                  = aux1 m a b xs

-- ---------------------------------------------------------------------
-- Ejercicio 1.4.2. Definir la función valoresExternos por filtrado,
-- ---------------------------------------------------------------------
```

```
valoresExternos2 :: [Double] -> [Double]
valoresExternos2 xs =
    filter p xs
    where p x = x<m-(b-a)/2 || x>m+(b-a)/2
          m   = media xs
          a   = minimum xs
          b   = maximum xs

-- ---------------------------------------------------------------------
-- Ejercicio 1.4.3. Definir la función valoresExternos por comprensión.
-- ---------------------------------------------------------------------

valoresExternos3 :: [Double] -> [Double]
valoresExternos3 xs =
    [x | x <- xs, x<m-(b-a)/2 || x>m+(b-a)/2]
    where m = media xs
          a = minimum xs
          b = maximum xs

-- ---------------------------------------------------------------------
-- Ejercicio 1.5. Comprobar con QuickCheck si se verifica alguna de las
-- siguientes propiedades:
-- * La media de una lista no vacía es mayor o igual que la mediana.
-- * La media de una lista no vacía es menor o igual que la mediana.
-- * La media de una lista no vacía es menor o igual que (minimo + maximo)/2
-- ---------------------------------------------------------------------

-- Las propiedades son:
prop1, prop2, prop3 :: [Double] -> Property
prop1 xs = not (null xs) ==> media xs >= mediana xs
prop2 xs = not (null xs) ==> media xs <= mediana xs
prop3 xs = not (null xs) ==> media xs <= (minimum xs + maximum xs)/2

-- La comprobación con QuickCheck es
--    Main> quickCheck prop1
--    Falsifiable, after 3 tests:
--    [3.0,3.0,2.66666666666667,-0.8]
--
--    Main> quickCheck prop2
--    Falsifiable, after 10 tests:
```

```
--    [-4.33333333333333,-3.75,-2.0,4.0]
--
--    Main> quickCheck prop3
--    Falsifiable, after 2 tests:
--    [1.75,-4.33333333333333,2.66666666666667,-3.2]
-- Por tanto, no se verifica ninguna de las propiedades.

-- ---------------------------------------------------------------------------
-- Ejercicio 2.1. Calcular el menor número natural que posee más de 20
-- divisores.
-- ---------------------------------------------------------------------------

número1 :: Integer
número1 = head [x | x <- [1..], númeroDiv x > 20]

númeroDiv :: Integer -> Int
númeroDiv n = length (divisores n)

divisores :: Integer -> [Integer]
divisores n = [x | x <- [1..n], n `mod` x == 0]

-- El cálculo es
--    Main> número1
--    360

-- ---------------------------------------------------------------------------
-- Ejercicio 2.2. Calcular 100-ésimo número primo.
-- ---------------------------------------------------------------------------

número2 :: Integer
número2 =  [x | x <- [1..], primo x] !! 99

primo :: Integer -> Bool
primo n = divisores n == [1,n]

-- El cálculo es
--    Main> número2
--    541

-- ---------------------------------------------------------------------------
```

```
-- Ejercicio 2.3. Calcular el menor primo de 4 cifras, p, tal que p y (p+2)
-- son primos.
-- ---------------------------------------------------------------------

número3 :: Integer
número3 = head [x | x <- [1000..9999],
                    primo x,
                    primo (x+2)]

-- El cálculo es
--    Main> número3
--    1019

-- ---------------------------------------------------------------------
-- Ejercicio 2.4. Calcular el mayor primo de 4 cifras, p, tal que p y (p+2)
-- son primos.
-- ---------------------------------------------------------------------

número4 :: Integer
número4 = head [x | x <- [9999,9998..1000],
                    primo x,
                    primo (x+2)]

-- El cálculo es
--    Main> número4
--    9929

-- ---------------------------------------------------------------------
-- Ejercicio 2.5. Calcular el menor número natural de la forma 2^(2^n) + 1 que
-- no es primo.
-- ---------------------------------------------------------------------

número5 :: Integer
número5 = head [2^(2^n) + 1 | n <- [1..],
                              not (primo (2^(2^n)+1))]

-- El cálculo es
--    Main> número5
--    4294967297
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 2.6. Calcular dos números primos a y b tales que (a^2+b^2+1) y
-- (a^3+b^3-1) son también números primos.
-- ---------------------------------------------------------------------

par_número6 :: (Integer,Integer)
par_número6 = head [(a,b) | a <- [2..],
                            b <- [2..a],
                            primo a,
                            primo b,
                            primo(a^2+b^2+1),
                            primo(a^3+b^3+1)]

-- El cálculo es
--    Main> par_número6
--    (31,3)

-- ---------------------------------------------------------------------
-- Ejercicio 3 (Eliminación de duplicados)
-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir la función duplicados 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
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir la función eliminaDuplicados 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]
-- ---------------------------------------------------------------------

-- 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 3.3. Comprobar con QuickCheck que siempre el valor de
-- eliminaDuplicados es una lista sin duplicados.
-- ------------------------------------------------------------------------

-- 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 3.4. ¿Se puede garantizar con la propiedad anterior que
-- eliminaDuplicados se comporta correctamente? En caso negativo, ¿qué
-- propiedad falta?
-- ---------------------------------------------------------------------

-- 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

-- ---------------------------------------------------------------------
-- Ejercicio 4. (Reconocimiento de permutaciones)
-- ---------------------------------------------------------------------
-- 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]. En este ejercicio vamos a
-- estudiar las permutaciones.
-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Definir la función esPermutación 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
-- ---------------------------------------------------------------------
```

```
-- 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 4.2. Comprobar con QuickCheck que si una lista es una permutación
-- de otra, las dos tienen el mismo número de elementos.
-- ---------------------------------------------------------------------

-- 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 4.3. Comprobar con QuickCheck que la inversa de una lista es una
-- permutación de la lista.
-- ---------------------------------------------------------------------

-- 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.
```

Capítulo 4

Práctica 2b (12 de noviembre de 2007)

4.1. Práctica 2b (12 de noviembre de 2007)

```
-- ---------------------------------------------------------------------
-- Importación de librerías auxiliares                                 --
-- ---------------------------------------------------------------------

import Test.QuickCheck
import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1 (Extracciones e inserciones)
-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir la función extrae tal que (extrae xs n) es la lista
-- resultado de eliminar el elemento n-ésimo de la lista xs. Si n es negativo
-- o mayor que la longitud de la lista el resultado debe ser la misma
-- lista. Por ejemplo,
--    extrae [1,2,3,4,5] 2     ==>  [1,2,4,5]
--    extrae [1,2,3,4,5] 4     ==>  [1,2,3,4]
--    extrae [1,2,3,4,5] (-1)  ==>  [1,2,3,4,5]
--    extrae [1,2,3,4,5] 7     ==>  [1,2,3,4,5]
-- ---------------------------------------------------------------------

-- Presentamos dos definiciones. La primera es
extrae1 :: [a] -> Int -> [a]
extrae1 xs n = (take n xs) ++ (drop (n+1) xs)

-- La segunda definición es recursiva
extrae2 :: [a] -> Int -> [a]
```

```
extrae2 xs     n | n<0     = xs
extrae2 []     _           = []
extrae2 (h:xs) 0           = xs
extrae2 (h:xs) (n+1)       = h:(extrae2 xs n)

-- En lo sucesivo, usaremos como definición la primera
extrae :: [a] -> Int -> [a]
extrae = extrae1

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir la función inserta tal que (inserta xs e n) debe
-- introducir el elemento n en la posición n de la lista xs. Si (n<=0) el
-- elemento se insertará al principio de la lista, y si n es igual o mayor que
-- la longitud de la lista, el elemento e se colocará al final de la lista. Por
-- ejemplo,
--    inserta [1,3,5] 7 1     ==>  [1,7,3,5]
--    inserta [1,3,5] 7 3     ==>  [1,3,5,7]
--    inserta [1,3,5] 7 (-1)  ==>  [7,1,3,5]
--    inserta [1,3,5] 7 9     ==>  [1,3,5,7]
-- ---------------------------------------------------------------------

-- Presentamos dos definiciones. La primera es
inserta1 :: [a] -> a -> Int -> [a]
inserta1 xs e n = (take n xs) ++ [e] ++ (drop n xs)

-- La segunda definición es recursiva
inserta2 :: [a] -> a -> Int -> [a]
inserta2 xs     e n | n<=0 = e:xs
inserta2 []     e _        = [e]
inserta2 (h:xs) e (n+1)    = h:(inserta2 xs e n)

-- En lo sucesivo, usaremos como definición la primera
inserta :: [a] -> a -> Int -> [a]
inserta = inserta2

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Comprobar con QuickCheck que si se inserta el elemento
-- n-ésimo de una lista xs en el resultado de extraer el elemento n-ésimo de xs
-- se obtiene la lista xs.
-- ---------------------------------------------------------------------
```

```
-- La propiedad es
prop_InsertaExtrae :: [Int] -> Int -> Property
prop_InsertaExtrae xs n =
    0<=n && n < length xs ==>
    inserta (extrae xs n) (xs!!n) n == xs

-- La comprobación es
--    Main> quickCheck prop_InsertaExtrae
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 2. (Listas y conjuntos)
-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Definir la función eliminaDuplicados 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]
-- ---------------------------------------------------------------------

-- 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

-- Nota: La función eliminaDuplicados es equivalente a la función nub de la
-- librería Data.List.

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir la función esConjunto tal que (esConjunto xs) se
-- verifica si la lista xs es un conjunto; es decir, no contiene elementos
-- duplicados, Por ejemplo,
--    esConjunto [1,2,3,4,5]  ==>  True
--    esConjunto [1,2,3,2]    ==>  False
-- ---------------------------------------------------------------------

esConjunto :: Eq a => [a] -> Bool
esConjunto []     = True
esConjunto (x:xs) = notElem x xs && esConjunto xs

-- ---------------------------------------------------------------------
-- Ejercicio 2.3. Comprobar con QuickCheck que siempre el valor de
-- eliminaDuplicados es un conjunto.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_duplicadosEliminados :: [Int] -> Bool
prop_duplicadosEliminados xs = esConjunto (eliminaDuplicados xs)
-- y la comprobación es
--    Main> quickCheck prop_duplicadosEliminados
```

```
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 3. (Operaciones y relaciones con conjuntos)
-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir la función borra tal que (borra x ys) es la lista
-- obtenida borrando la primera ocurrencia del elemento x en la lista xs. Por
-- ejemplo,
--    borra 'a' "Salamanca"  ==>  "Slamanca"
-- ---------------------------------------------------------------------

borra :: Eq a => a -> [a] -> [a]
borra _ []     = []
borra x (y:ys) = if x == y then ys else (y : borra x ys)

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Comprobar con QuickCheck que la función borra es equivalente
-- a la función delete de la librería Data.List.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_borraEquivDelete :: Int -> [Int] -> Bool
prop_borraEquivDelete x ys =
    borra x ys == delete x ys

-- y su comprobación es
--    Main> quickCheck prop_borraEquivDelete
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. Definir la relación subconjunto tal que (subconjunto xs ys)
-- se verifica si todos los elementos de xs pertenecen a ys. Por ejemplo,
--    subconjunto "abac" "cba"   ==>  True
--    subconjunto "abacd" "cba"  ==>  False
-- ---------------------------------------------------------------------

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

```
-- ---------------------------------------------------------------------
-- Ejercicio 3.4. Definir el operador &= tal que (xs &= ys) se verifique si las
-- listas xs e ys son iguales como conjuntos; es decir cada una es un sub
-- conjunto de la otra. Por ejemplo,
--    [1,2,3] &= [3,2,1]      ==>  True
--    [1,2,3] &= [1,2]        ==>  False
--    [1,2,3] &= [1,2,3,2,1]  ==>  True
-- ---------------------------------------------------------------------

infix 4 &=
(&=) :: Eq a => [a] -> [a] -> Bool
xs &= ys = subconjunto xs ys && subconjunto ys xs

-- ---------------------------------------------------------------------
-- Ejercicio 3.5. Definir la función unión tal que (unión xs ys) es el
-- conjunto de los elementos que pertenecen a xs o a ys. Por ejemplo,
--    unión "ab" "bc"    ==>  "abc"
--    unión "ab" "bcc"   ==>  "abc"
--    unión "aba" "bcc"  ==>  "abc"
-- ---------------------------------------------------------------------

unión [] ys                    = nub ys
unión (x:xs) ys
    | (elem x xs || elem x ys) = unión xs ys
    | otherwise                = x : unión xs ys

-- ---------------------------------------------------------------------
-- Ejercicio 3.6. Comprobar con QuickCheck que la unión siempre devuelve un
-- conjunto.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_UniónEsConjunto :: [Int] -> [Int] -> Bool
prop_UniónEsConjunto xs ys =
    esConjunto (unión xs ys)

-- y su comprobación es
--    Main> quickCheck prop_UniónEsConjunto
--    OK, passed 100 tests.
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 3.7. Comprobar con QuickCheck las siguientes propiedades de la
-- unión:
--  * xs está contenido en la unión de xs con ys.
--  * ys está contenido en la unión de xs con ys.
--  * si xs e ys están contenido en zs, entonces la unión de xs e ys está
--    contenido en zs.
-- ---------------------------------------------------------------------

-- Las propiedades son
prop_Unión1 :: [Int] -> [Int] -> Bool
prop_Unión1 xs ys =
    subconjunto xs (unión xs ys)

prop_Unión2 :: [Int] -> [Int] -> Bool
prop_Unión2 xs ys =
    subconjunto ys (unión xs ys)

prop_Unión3 :: [Int] -> [Int] -> [Int] -> Property
prop_Unión3 xs ys zs =
    subconjunto xs zs && subconjunto ys zs ==>
    subconjunto (unión xs ys) zs

-- y sus comprobaciones son
--    Main> quickCheck prop_Unión1
--    OK, passed 100 tests.
--
--    Main> quickCheck prop_Unión2
--    OK, passed 100 tests.
--
--    Main> quickCheck prop_Unión3
--    Arguments exhausted after 89 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 3.8. En la librería Data.List está definida la función
-- union. Comprobar con QuickCheck si es equivalente a la función unión que
-- hemos definido. En el caso que no lo sea, establecer una relación entre las
-- dos funciones y comprobarla con QuickCheck.
-- ---------------------------------------------------------------------
```

```
-- La propiedad que establece que las definiciones son equivalentes es
prop_UniónEquivUnion :: [Int] -> [Int] -> Bool
prop_UniónEquivUnion xs ys =
    unión xs ys == union xs ys

-- Su comprobación es
--    Main> quickCheck prop_UniónEquivUnion
--    Falsifiable, after 6 tests:
--    [-3,-3,-4,-1]
--    [-3,-3,-4]
-- que indica que no son equivalentes.

-- La propiedad que se verifica es que sus valores son iguales como conjuntos
prop_UniónEquivUnion2 :: [Int] -> [Int] -> Bool
prop_UniónEquivUnion2 xs ys =
    unión xs ys &= union xs ys

-- En efecto,
--    Main> quickCheck prop_UniónEquivUnion2
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 3.9. Definir la función intersección tal que (intersección xs ys)
-- es el conjunto de los elementos comunes de las listas xs e ys. Por ejemplo,
--    intersección [1,2,3] [1,2,3]        ==>  [1,2,3]
--    intersección [1,3,5,7] [3,4,5]      ==>  [3,5]
--    intersección [1,3,5,1,7] [3,4,5,3]  ==>  [3,5]
-- ---------------------------------------------------------------------

intersección :: Eq a => [a] -> [a] -> [a]
intersección xs ys = [a | a <- (nub xs), elem a ys]

-- ---------------------------------------------------------------------
-- Ejercicio 3.10. Comprobar con QuickCheck que la intersección siempre
-- devuelve un conjunto.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_IntersecciónEsConjunto :: [Int] -> [Int] -> Bool
prop_IntersecciónEsConjunto xs ys =
```

```
    esConjunto (intersección xs ys)

-- y su comprobación es
--    Main> quickCheck prop_IntersecciónEsConjunto
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 3.11. Comprobar con QuickCheck las siguientes propiedades de la
-- intersección:
--  * la intersección de xs con ys está contenido en xs
--  * la intersección de xs con ys está contenido en ys
--  * si zs está contenido en xs e ys, entonces zs está contenido en la
--    intersección de xs e ys.
-- ---------------------------------------------------------------------

-- Las propiedades son
prop_Intersección1 :: [Int] -> [Int] -> Bool
prop_Intersección1 xs ys =
    subconjunto (intersección xs ys) xs

prop_Intersección2 :: [Int] -> [Int] -> Bool
prop_Intersección2 xs ys =
    subconjunto (intersección xs ys) ys

prop_Intersección3 :: [Int] -> [Int] -> [Int] -> Property
prop_Intersección3 xs ys zs =
    subconjunto zs xs && subconjunto zs ys ==>
    subconjunto zs (intersección xs ys)

-- y sus comprobaciones son
--    Main> quickCheck prop_Intersección1
--    OK, passed 100 tests.
--
--    Main> quickCheck prop_Intersección2
--    OK, passed 100 tests.
--
--    Main> quickCheck prop_Intersección3
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
```

```
-- Ejercicio 3.12. En la librería Data.List está definida la función
-- intersect. Comprobar con QuickCheck si es equivalente a la función
-- intersección que hemos definido. En el caso que no lo sea, establecer una
-- relación entre las dos funciones y comprobarla con QuickCheck.
-- ------------------------------------------------------------------------

-- La propiedad que establece que las definiciones son equivalentes es
prop_IntersecciónEquivIntersect :: [Int] -> [Int] -> Bool
prop_IntersecciónEquivIntersect xs ys =
    intersección xs ys == intersect xs ys

-- Su comprobación es
--    Main> quickCheck prop_IntersecciónEquivIntersect
--    Falsifiable, after 7 tests:
--    [5,5,1,4,1]
--    [5]
-- que indica que no son equivalentes.

-- La propiedad que se verifica es que sus valores on iguales como conjuntos
prop_IntersecciónEquivIntersect2 :: [Int] -> [Int] -> Bool
prop_IntersecciónEquivIntersect2 xs ys =
    intersección xs ys &= intersect xs ys

-- En efecto,
--    Main> quickCheck prop_IntersecciónEquivIntersect2
--    OK, passed 100 tests.

-- ------------------------------------------------------------------------
-- Ejercicio 3.13. Definir la función diferencia tal que (diferencia xs ys) es
-- el conjunto de los elementos de xs que no pertenecen a ys. Por ejemplo,
--    diferencia [1,2,3,4,5] [1,3,5]           ==>  [2,4]
--    diferencia [1,3,5,3,7,1] [2,4,3,6,3,5,8]  ==>  [1,7]
--    diferencia "sevilla" "betis"             ==>  "vla"
-- ------------------------------------------------------------------------

diferencia :: Eq a => [a] -> [a] -> [a]
diferencia xs ys = [a | a <- (nub xs), notElem a ys]

-- ------------------------------------------------------------------------
-- Ejercicio 3.14. Comprobar con QuickCheck que la diferencia siempre devuelve
```

```
-- un conjunto.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_DiferenciaEsConjunto :: [Int] -> [Int] -> Bool
prop_DiferenciaEsConjunto xs ys =
    esConjunto (diferencia xs ys)

-- y su comprobación es
--    Main> quickCheck prop_DiferenciaEsConjunto
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 3.15. Comprobar con QuickCheck las siguientes propiedades de la
-- diferencia:
--  * la diferencia de xs con ys está contenido en xs
--  * la diferencia de xs con ys es disjunta con ys; es decir la intersección
--    entre la diferencia de xs e ys con ys es el conjunto vacío.
-- ---------------------------------------------------------------------

-- Las propiedades son
prop_Diferencia1 :: [Int] -> [Int] -> Bool
prop_Diferencia1 xs ys =
    subconjunto (diferencia xs ys) xs

prop_Diferencia2 :: [Int] -> [Int] -> Bool
prop_Diferencia2 xs ys =
     intersección (diferencia xs ys) ys == []

-- y sus comprobaciones son
--    Main> quickCheck prop_Diferencia1
--    OK, passed 100 tests.
--
--    Main> quickCheck prop_Diferencia2
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 3.16. En la librería Data.List está definido el operador
-- (\\). Comprobar con QuickCheck si es equivalente a la función
-- intersección que hemos definido. En el caso que no lo sea, establecer una
```

```
-- relación entre las dos funciones y comprobarla con QuickCheck.
-- ---------------------------------------------------------------------

-- La propiedad que establece que las definiciones son equivalentes es
prop_DiferenciaEquiv :: [Int] -> [Int] -> Bool
prop_DiferenciaEquiv xs ys =
    diferencia xs ys == xs \\ ys

-- Su comprobación es
--    Main> quickCheck prop_DiferenciaEquiv
--    Falsifiable, after 10 tests:
--    [1,0,-1,-7,8,-6,1,-6]
--    [0,0]
-- que indica que no son equivalentes.

-- La propiedad que se verifica es que la diferencia de xs y ys es igual como
-- conjunto a la diferencia entre que el conjunto correspondiente a xs y
-- la lista ys.
prop_DiferenciaEquiv2 :: [Int] -> [Int] -> Bool
prop_DiferenciaEquiv2 xs ys =
    diferencia xs ys &= (nub xs) \\ ys

-- En efecto,
--    Main> quickCheck prop_DiferenciaEquiv2
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 3.17. Comprobar con QuickCheck si la unión de xs e ys es igual,
-- como conjunto, que la concatenación de conjunto correspondiente a xs y su
-- diferencia con ys.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_UniónConcatenaciónDiferencia :: [Int] -> [Int] -> Bool
prop_UniónConcatenaciónDiferencia xs ys =
    unión xs ys &= (nub xs) ++ (diferencia (nub ys) xs)

-- La comprobación es
--    Main> quickCheck prop_UniónConcatenaciónDiferencia
--    OK, passed 100 tests.
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 4. (El triángulo de Pascal)
-- ---------------------------------------------------------------------
-- Ejercicio 4. 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 4.1. Definir la función pascal tal que (pascal n) es la
-- n-ésima fila del triángulo de Pascal. Por ejemplo,
--    pascal 6  ==>  [1,5,10,10,5,1]
-- ---------------------------------------------------------------------

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 4.2. Comprobar con QuickCheck, que la fila n-ésima del triángulo
-- de Pascal tiene n elementos.
-- ---------------------------------------------------------------------

-- 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

-- La comprobación es
--    Main> quickCheck prop_Pascal
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 4.3. Comprobar con QuickCheck, que la suma de los elementos de la
-- fila n-ésima del triángulo de Pascal es igual a 2^(n-1).
-- ---------------------------------------------------------------------

-- la propiedad es
prop_sumaPascal :: Integer -> Property
prop_sumaPascal n =
    n >= 1 ==> sum (pascal n) == 2^(n-1)

-- La comprobación es
--    Main> quickCheck prop_sumaPascal
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 4.4. Comprobar con QuickCheck, que el m-ésimo elemento de la fila
-- (n+1)-ésima del triángulo de Pascal es el número combinatorio
-- (n sobre m) = n! / (k!(n-k)!).
-- ---------------------------------------------------------------------
```

```
--  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 y (comb n k) es el número combinatorio n
-- sobre k.
fact :: Integer -> Integer
fact n = product [1..n]

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.
```

Capítulo 5

Examen 1a (19 de Noviembre de 2007)

5.1. Examen 1 de gupos 1A y 1B

```
-- ---------------------------------------------------------------------
-- Importación de librerías auxiliares
-- ---------------------------------------------------------------------

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1 (Puntos en círulo)
-- ---------------------------------------------------------------------
-- Definir la función
--    puntosEnCírculo :: Double -> [(Double,Double)] -> [(Double,Double)]
-- tal que (puntosEnCírculo r xs) es la lista de puntos xs que se
-- encuentra en el círculo de centro (0,0) y radio r. Por ejemplo,
--    Main> puntosEnCírculo 1 [(0,0),(0.5,0.5),(1,0),(0,1),(1,1)]
--    [(0.0,0.0),(0.5,0.5),(1.0,0.0),(0.0,1.0)]
-- ---------------------------------------------------------------------

puntosEnCírculo :: Double -> [(Double,Double)] -> [(Double,Double)]
puntosEnCírculo r xs = [ (x,y) | (x,y) <- xs, sqrt (x^2 + y^2) <= r ]

-- ---------------------------------------------------------------------
-- Ejercicio 2 (Lista de asociación)
-- ---------------------------------------------------------------------
-- Se puede representar la cantidad de elementos de distintas clases
-- mediante una lista de asociación. Por ejemplo, la lista
-- [('a',50),('b',20),('c',8)] indica que existen 50 elementos de tipo
```

```
-- a, 20 de tipo b y 8 de tipo c.
-- ---------------------------------------------------------------------
-- Ejercicio 2.1 Definir la función
--    elementos :: Eq a => [(a,b)] -> [a]
-- tal que (elementos xs) es la lista de elementos en la lista de
-- asociación xs. Por ejemplo,
--    elementos [('a',5),('b',2),('c',4),('a',3)]  ==>  "abc"
-- ---------------------------------------------------------------------

elementos :: Eq a => [(a,b)] -> [a]
elementos [] = []
elementos ((x,v):xs) = x:(elementos [ (y,v) | (y,v) <- xs, y /= x ])

-- ---------------------------------------------------------------------
-- Ejercicio 2.2 Definir la función
--    valor :: Eq b => b -> [(b,Int)] -> Int
-- tal que (valor x ys) es la suma de objetos de la clase x en la lista
-- de asociación ys. Por ejemplo,
--    valor 'a' [('a',5),('b',2),('a',7)]  ==>  12
-- ---------------------------------------------------------------------

valor :: Eq b => b -> [(b,Int)] -> Int
valor a xs = sum [ v | (x,v) <- xs, x == a ]

-- ---------------------------------------------------------------------
-- Ejercicio 2.3 Definir la función
--    elimina :: Eq a => a -> [(a,Int)] -> [(a,Int)]
-- tal que (elimina x ys) es el resultado de eliminar los objetos de la
-- clase x en la lista ys. Por ejemplo,
--    elimina 'a' [('a',5),('b',2),('a',7)]  ==>  [('b',2)]
-- ---------------------------------------------------------------------

elimina :: Eq a => a -> [(a,Int)] -> [(a,Int)]
elimina a xs = [ (x,v) | (x,v) <- xs, x /= a ]

-- ---------------------------------------------------------------------
-- Ejercicio 2.4 Definir la función
--    unión :: Eq a => [(a,Int)] -> [(a,Int)] -> [(a,Int)]
-- tal que (unión xs ys) es la lista de asociación cuyos elementos son
-- los de xs o ys y el número de ocurrencias es la suma de las
```

```
-- courrencias en xs y en ys. Por ejemplo:
--    Main> unión [('a',5),('b',2)] [('b',4),('a',3)]
--    [('a',8),('b',6)]
--    Main> unión [('a',5),('b',2)] [('b',4),('a',3),('c',7)]
--    [('a',8),('b',6),('c',7)]
--    Main> unión [('a',5),('b',2)] [('b',4),('a',3),('b',7)]
--    [('a',8),('b',13)]
--    Main> unión [('a',5),('b',2),('a',2)] [('b',4),('a',3),('b',7)]
--    [('a',10),('b',13)]
-- Nótese que se debe considerar el caso en el que en las listas
-- pudiera estar repetida más de una vez la misma clase (como en los
-- ejemplos tercero y cuarto).
-- ---------------------------------------------------------------------

unión :: Eq a => [(a,Int)] -> [(a,Int)] -> [(a,Int)]
unión xs ys = [ (x, valor x zs) | x <- elementos zs  ]
              where zs = xs ++ ys
```

5.2. Examen 1 de gupos 2A y 2B

```
-- ---------------------------------------------------------------------
-- Importación de librerías auxiliares
-- ---------------------------------------------------------------------

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1 (Separación de listas)
-- ---------------------------------------------------------------------
-- Definir la función
--    separa :: [Int] -> ([Int],[Int])
-- que separa una lista de números enteros en sus elementos pares e
-- impares. Por ejemplo,
--    separa [1,2,3,4,5]  ==>  ([2,4],[1,3,5])
--    separa [1,3,7,1]    ==>  ([],[1,3,7,1])
-- ---------------------------------------------------------------------

-- Presentamos tres definiciones.

-- La primera defnición es recursiva:
```

```
separa1 :: [Int] -> ([Int],[Int])
separa1 [] = ([],[])
separa1 (x:xs)
    | x `mod` 2 == 0 = (x:ys,zs)
    | otherwise      = (ys,x:zs)
    where (ys,zs) = separa1 xs

-- La segunda definición es por comprensión
separa2 :: [Int] -> ([Int],[Int])
separa2 xs = ( [ x | x <- xs, x `mod` 2 == 0 ],
               [ x | x <- xs, x `mod` 2 == 1 ] )

-- La tercera definición es por filtrado
separa3 :: [Int] -> ([Int],[Int])
separa3 xs = (filter par xs, filter impar xs)
    where par x = x `mod` 2 == 0
          impar x = not (par x)

-- ---------------------------------------------------------------------
-- Ejercicio 2 (Repeticiones)
-- ---------------------------------------------------------------------
-- Ejercicio 2.1 Definir recursivamente la función
--    repite :: Int -> String -> String
-- tal que (repite n cs) es la cadena obtenida repetiendo n veces cada
-- caracter de cs. Por ejemplo,
--    repite 3 "Haskell"    => "HHHaaassskkkeeellllll"
--    repite 2 "Hola mundo" => "HHoollaa  mmuunnddoo"
--    repite (-1) "hola"    => ""
-- ---------------------------------------------------------------------

repite1 :: Int -> String -> String
repite1 _ "" = ""
repite1 n _ | n <= 0 = ""
repite1 n (c:cs) = (replicate n c) ++ repite1 n cs

-- ---------------------------------------------------------------------
-- Ejercicio 2.2 Definir la función repite usando listas por
-- comprensión.
-- ---------------------------------------------------------------------
```

```
repite2 :: Int -> String -> String
repite2 n cs = [c | c <- cs, i <- [1..n]]

-- ---------------------------------------------------------------------
-- Ejercicio 3 (Grafos completos)
-- ---------------------------------------------------------------------
-- Ejercicio 3 Se define un tipo de dato polimórfico (Grafo a) para
-- representar grafos no dirigidos mediante la lista de sus nodos y la
-- listas de sus aristas como sigue:
--    data Grafo a = G [a] [(a,a)]
--                   deriving Show
-- De esta manera, la representación del grafo g1
--    1 -- 2 -- 3  5
--         |   /
--         |  /
--         | /
--         |/
--         4
-- es la siguiente
--    g1 :: Grafo Int
--    g1 = G [1,2,3,4,5] [(1,2),(2,3),(2,4),(3,4)]
-- ---------------------------------------------------------------------

data Grafo a = G [a] [(a,a)]
               deriving Show

g1 :: Grafo Int
g1 = G [1,2,3,4,5] [(1,2),(2,3),(2,4),(3,4)]

-- ---------------------------------------------------------------------
-- Ejercicio 3.1 Definir la función
--    grado :: Eq a => Grafo a -> a -> Int
-- tal que (grado g x) es el grado del nodo x en el grafo g; esto es, el
-- número de aristas del grafo en las que aparece el nodo x. Si x no es
-- un nodo del grafo, entonces la función debe devolver (-1). Por
-- ejemplo,
--    grado g1 2  ==>  3
--    grado g1 3  ==>  2
--    grado g1 5  ==>  0
--    grado g1 7  ==>  -1
```

```
-- ---------------------------------------------------------------------

grado :: Eq a => Grafo a -> a -> Int
grado (G xs ys) x
    | elem x xs = length([n | (n,m) <- ys, m==x]++[n | (m,n) <- ys, x==m])
    | otherwise = (-1)

-- ---------------------------------------------------------------------
-- Ejercicio 3.2 Definir la función
--    grafoCompleto :: Int -> Grafo Int
-- tal que (grafoCompleto n) es el grafo completo de nivel n, esto es,
-- un grafo con n nodos [1,2,..,n] y tal que dos nodos cualesquiera
-- están unidos por una arista. Por ejemplo,
--    grafoCompleto 3 => G [1,2,3] [(1,2),(1,3),(2,3)]
--    grafoCompleto 4 => G [1,2,3,4] [(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]
-- ---------------------------------------------------------------------

grafoCompleto :: Int -> Grafo Int
grafoCompleto n = G [1..n] [(a,b) | a <- [1..n], b <- [(a+1)..n]]
```

Capítulo 6

Examen 1b (26 de Noviembre de 2007)

6.1. Examen 1 de gupos 1C y 1D

```
-- ---------------------------------------------------------------------
-- Importación de librerías auxiliares
-- ---------------------------------------------------------------------

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1 (Diferencias de cuadrados)
-- ---------------------------------------------------------------------
-- Ejercicio 1.1 Definir la función
--    sucesiónImpares :: Int -> [Int]
-- tal que (sucesiónImpares n) es la sucesión de los n primeros números
-- impares. Por ejemplo,
--    sucesiónImpares 5 ==> [1,3,5,7,9]
-- ---------------------------------------------------------------------

sucesiónImpares :: Int -> [Int]
sucesiónImpares n = [2*x-1 | x <- [1..n]]

-- ---------------------------------------------------------------------
-- Ejercicio 1.2 Definir la función
--    cuadrados :: Int -> [Int]
-- tal que (cuadrados n) es la sucesión de los cuadrados de los números
-- entre 1 y n. Por ejemplo,
--    cuadrados 5 ==> [1,4,9,16,25]
-- ---------------------------------------------------------------------
```

```
cuadrados :: Int -> [Int]
cuadrados n = [a^2 | a <- [1..n]]

-- ---------------------------------------------------------------------
-- Ejercicio 1.3 Definir la función
--   restaCuadrados:: Int -> [Int]
-- tal que (restaCuadrados n) es la lista obtenida restando los
-- cuadrados consecutivos de los números entre 1 y n. Por ejemplo,
--    restaCuadrados 5 ==> [3,5,7,9]
-- ---------------------------------------------------------------------

restaCuadrados:: Int -> [Int]
restaCuadrados n = restaSucesiones (cuadrados n)

-- donde (restaSucesiones xs) es la lista obtenida restando los
-- elementos consecutivos de la sucesión xs. Por ejemplo,
--   restaSucesiones [1,3,4,7] ==> [2,1,3]
restaSucesiones:: [Int] -> [Int]
restaSucesiones [x]      = []
restaSucesiones (x:y:xs) = (y-x):(restaSucesiones (y:xs))

-- ---------------------------------------------------------------------
-- Ejercicio 2 (Grafos eulerianos)
-- ---------------------------------------------------------------------
-- Se define un tipo de dato polimórfico (Grafo a) para
-- representar grafos dirigidos mediante la lista de sus nodos y la
-- función que asigna a cada nodo la lista de sus sucesores.
--    data Grafo a = G [a] (a -> [a])
-- De esta manera, la representación del grafo g1
--         1          5
--        / \
--       /   \
--      2     3
--       \   /
--        \ /
--         4
-- es la siguiente
--    g1 :: Grafo Int
--    g1 = G [1..5] suc
```

```
--         where suc 1 = [2,3]
--               suc 2 = [4]
--               suc 3 = [4]
--               suc 4 = []
-- y la del grafo g2
--    1 --> 2
--    ^     |
--    |     v
--    4 <-- 3
-- es la siguiente
--    g2 :: Grafo Int
--    g2 = G [1..4] suc
--         where suc 4 = [1]
--               suc x = [x+1]
-- ------------------------------------------------------------------------

data Grafo a = G [a] (a -> [a])

g1 :: Grafo Int
g1 = G [1..5] suc
     where suc 1 = [2,3]
           suc 2 = [4]
           suc 3 = [4]
           suc _ = []

g2 :: Grafo Int
g2 = G [1..4] suc
     where suc 4 = [1]
           suc x = [x+1]

-- ------------------------------------------------------------------------
-- Ejercico 2.1 Definir la función
--    anteriores :: Eq a => a -> (Grafo a) -> [a]
-- tal que (anteriores x g) es la lista de los anteriores del nodo x en
-- el grafo g. Por ejemplo,
--    anteriores 4 g1  ==>  [2,3]
--    anteriores 1 g1  ==>  []
--    anteriores 1 g2  ==>  [4]
-- ------------------------------------------------------------------------
```

```
anteriores :: Eq a => a -> (Grafo a) -> [a]
anteriores x (G vs s) = [y | y <- vs, elem x (s y)]

-- ---------------------------------------------------------------------
-- Ejercicio 2.2 Un grafo dirigido conexo es euleriano si todos los
-- nodos tienen el mismo número de sucesores que antecesores. Definir la
-- función
--    euleriano :: Eq a => (Grafo a) -> Bool
-- tal que (euleriano g) se verifica si grafo dirigido conexo g es
-- euleriano. Por ejemplo,
--    euleriano g1  ==>  False
--    euleriano g2  ==>  True
-- ---------------------------------------------------------------------

euleriano :: Eq a => (Grafo a) -> Bool
euleriano g@(G xs s) =
    and [length (anteriores x g) == length (s x) | x <- xs]
```

6.2. Examen 1 de gupos 2C y 2D

```
-- ---------------------------------------------------------------------
-- Importación de librerías auxiliares
-- ---------------------------------------------------------------------

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1 (Divisores)
-- ---------------------------------------------------------------------
-- Ejercicio 1.1 Definir recursivamente la función
--    múltiploDeTodos :: Int -> [Int] -> Bool
-- tal que (múltiploDeTodos x ys) se verifica si x es un múltiplo de
-- todos los elementos de ys. Por ejemplo,
--    múltiploDeTodos 12 [2,3]  ==>  True
--    múltiploDeTodos 14 [2,3]  ==>  False
-- ---------------------------------------------------------------------

múltiploDeTodos :: Int -> [Int] -> Bool
múltiploDeTodos x []     = True
múltiploDeTodos x (y:ys) = (múltiplo x y) && (múltiploDeTodos x ys)
```

```
-- donde (múltiplo x y) se verifica si x es un múltiplo de y. Por
-- ejemplo,
--    múltiplo 12 3  ==>  True
--    múltiplo 14 3  ==>  False
múltiplo :: Int -> Int -> Bool
múltiplo x y = x `mod` y == 0

-- ---------------------------------------------------------------------
-- Ejercicio 1.2 Definir la función
--    divisores :: Int -> Int -> [Int] -> [Int]
-- tal que (divisores a b xs) es la lista de los números
-- del intervalo [a..b] que son múltiplos de todos los elementos de la
-- lista xs. Por ejemplo,
--    divisores 10 20 [2,3]        ==>  [12,18]
--    divisores 1 50 [5]           ==>  [5,10,15,20,25,30,35,40,45,50]
--    divisores 1 100 [1,2,3,4,5]  ==>  [60]
--    divisores 2 9 []             ==>  [2,3,4,5,6,7,8,9]
--    divisores 1 100 [1..10]      ==>  []
-- ---------------------------------------------------------------------

divisores :: Int -> Int -> [Int] -> [Int]
divisores a b xs = [x | x <- [a..b], múltiploDeTodos x xs]

-- ---------------------------------------------------------------------
-- Ejercicio 2 (Resumen de listas)
-- ---------------------------------------------------------------------
-- Ejercicio 2.1 Definir la función
--    elimina :: Eq a => a -> [a] -> [a]
-- tal que (elimina x xs) es la lista obtenida eliminando en xs las
-- ocurrencias de x. Por ejemplo,
--    elimina 'a' "abaacdaf"  ==>  "bcdf"
-- ---------------------------------------------------------------------

elimina :: Eq a => a -> [a] -> [a]
elimina x xs = [y | y <- xs, y /= x]

-- ---------------------------------------------------------------------
-- Ejercicio 2.2 Definir la función
--    cuenta :: Eq a => a -> [a] -> Int
```

```
-- tal que (cuenta x ys) es el número de ocurrencias del elemento x en
-- la lista xs. Por ejemplo,
--    cuenta 'a' "abaacdaf"  ==>  4
-- ---------------------------------------------------------------------

cuenta :: Eq a => a -> [a] -> Int
cuenta x xs = length [y | y <- xs, y == x]

-- ---------------------------------------------------------------------
-- Ejercicio 2.3 Definir la función
--    resumenLista :: Eq a => [a] -> [(a,Int)]
-- tal que (resumenLista xs) es la lista de pares donde cada par tiene
-- como primer elemento un elemento de xs y el segundo elemento del par
-- es el número de veces que dicho elemento se repite en la lista. Por
-- ejemplo,
--    resumenLista [1,2,3,1,2,3]  ==>  [(1,2),(2,2),(3,2)]
--    resumenLista "aabcadcd"     ==>  [('a',3),('b',1),('c',2),('d',2)]
-- ---------------------------------------------------------------------

resumenLista :: Eq a => [a] -> [(a,Int)]
resumenLista []     = []
resumenLista (x:xs) =
    (x,cuenta x (x:xs)):resumenLista (elimina x xs)
```

Capítulo 7

Práctica 3a (3 de diciembre de 2007)

7.1. Práctica 3a en Prolog

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 1 (Relaciones familiares)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% En este ejercicio se considera la siguiente base de conocimiento que
% describe algunas relaciones familiares mediante la relación
% padre(X,Y), que se verifica si X es padre de Y, y madre(X,Y), que se
% verifica si X es madre de Y.
padre(andres,bernardo).
padre(andres,belen).
padre(andres,baltasar).
padre(baltasar,david).
padre(david,emilio).
padre(emilio,francisco).
madre(ana,bernardo).
madre(ana,belen).
madre(ana,baltasar).
madre(belen,carlos).
madre(belen,carmen).
% Se pide:
% (a) Extender la base de conocimiento anterior mediante reglas para
%     definir las siguientes relaciones familiares:
%     (a.1) progenitor(X,Y) que se verifica si X es progenitor de Y (es
%           decir, X es el padre o la madre de Y),
%     (a.2) abuelo(X,Y) que se verifica si X es abuelo de Y.
% (b) Usar las relaciones definidas para responder a las siguientes
%     cuestiones:
```

```
%     (b.1) ¿quiénes son los progenitores de Bernardo?
%     (b.2) ¿quién es el abuelo de Carlos?
%     (b.3) ¿quiénes son los hijos de Andrés?
%     (b.4) ¿quiénes son los hijos de algún hijo de Ana?
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% (a.1) Definición de progenitor(X,Y)
progenitor(X,Y) :-
   padre(X,Y).
progenitor(X,Y) :-
   madre(X,Y).

% (a.2) Definición de abuelo(X,Y)
abuelo(X,Y) :-
   padre(X,Z),
   progenitor(Z,Y).

% (b.1) ¿quiénes son los progenitores de bernardo?
%    ?- progenitor(X,bernardo).
%    X = andrés ;
%    X = ana ;
%    No

% (b.2) ¿quién es el abuelo de carlos?
%    ?- abuelo(X,carlos).
%    X = andrés ;
%    No

% (b.3) ¿quiénes son los hijos de Andrés?
%    ?- padre(andres,X).
%    X = bernardo ;
%    X = belen ;
%    X = baltasar ;
%    No

% (b.4) ¿quiénes son los hijos de algún hijo de Ana?
%    ?- madre(ana,_X), progenitor(_X,Y).
%    Y = carlos ;
%    Y = carmen ;
%    Y = david ;
```

```
%    No

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 2 (Números naturales)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% En este ejercicio se usa la representación de los números naturales
% construidos con 0 y s; es decir, la serie
%    0, s(0), s(s(0)), s(s(s(0))), ...
% representa a la serie de los números naturales
%    0, 1, 2, 3, ...
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 2.1 Definir la relación suma(X,Y,Z) que se verifique si Z es
% la suma de los números naturales X e Y. Por ejemplo,
%    ?- suma(s(s(0)),s(s(s(0))),X).
%    X = s(s(s(s(0)))) ;
%    No
%    ?- suma(s(s(0)),X,s(s(s(s(s(0)))))).
%    X = s(s(0)) ;
%    No
%    ?- suma(X,s(s(s(0))),s(s(s(s(s(0)))))).
%    X = s(s(0)) ;
%    No
%    ?- suma(X,Y,s(s(s(s(s(0)))))).
%    X = 0
%    Y = s(s(s(s(0)))) ;
%    X = s(0)
%    Y = s(s(s(0)))
%    Yes
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

suma(0,Y,Y).
suma(s(X),Y,s(Z)) :-
   suma(X,Y,Z).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 2.2 Definir la relación menor(X,Y) que se verifique si el
% número natural X es menor que el número natural Y. Por ejemplo,
%    ?- menor(s(0),s(s(0))).
%    Yes
%    ?- menor(s(s(0)),s(0)).
```

```
%    No
%    ?- menor(s(0),s(0)).
%    No
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

menor(0,s(_Y)).
menor(s(X),s(Y)) :-
   menor(X,Y).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 2.2. Definir la relación mayor(X,Y) que se verifique si el
% número natural X es mayor que el número natural Y. Por ejemplo,
%    ?- mayor(s(0),s(s(0))).
%    No
%    ?- mayor(s(s(0)),s(0)).
%    Yes
%    ?- mayor(s(0),s(0)).
%    No
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

mayor(X,Y) :-
   menor(Y,X).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 2.3. Definir la relación resta(X,Y,Z) que se verifique si Z
% es X menos Y si X es mayor o igual que Y y 0, en caso contrario. Por
% ejemplo,
%    ?- resta(s(s(s(0))),s(0),X).
%    X = s(s(0)) ;
%    No
%    ?- resta(s(s(s(0))),s(s(0)),X).
%    X = s(0) ;
%    No
%    ?- resta(s(s(s(0))),s(s(s(0))),X).
%    X = 0 ;
%    No
%    ?- resta(s(s(s(0))),s(s(s(s(0)))),X).
%    X = 0 ;
%    No
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
resta(X,Y,Z) :-
   mayor(X,Y),
   suma(Y,Z,X).
resta(X,X,0).
resta(X,Y,0) :-
   menor(X,Y).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 2.4 Definir la relación producto(X,Y,Z) que se verifique si
% Z es el producto de los números naturales X e Y. Por ejemplo,
%    ?- producto(s(s(0)),s(s(s(0))),X).
%    X = s(s(s(s(s(0))))) ;
%    No
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

producto(0,_Y,0).
producto(s(X),Y,Z) :-
   producto(X,Y,Z1),
   suma(Y,Z1,Z).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 2.5 Definir la relación cuadrado(X,Y) que se verifique si Y
% es el cuadrado de X. Por ejemplo,
%    ?- cuadrado(s(s(0)),X).
%    X = s(s(s(s(0)))) ;
%    No
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

cuadrado(X,Y) :-
   producto(X,X,Y).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 2.6 Definir la relación cubo(X,Y) que se verifique si Y es
% el cubo de X. Por ejemplo,
%    ?- cubo(s(s(0)),X).
%    X = s(s(s(s(s(s(s(s(0))))))))) ;
%    No
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
cubo(X,Y) :-
   cuadrado(X,Y1),
   producto(X,Y1,Y).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 2.7 Definir la relación potencia(X,Y,Z) que se verifique si
% Z es la potencia Y-ésima de X. Por ejemplo,
%    ?- potencia(s(s(0)),s(s(s(0))),X).
%    X = s(s(s(s(s(s(s(s(0)))))))) ;
%    No
%    ?- potencia(s(s(s(0))),s(s(0)),X).
%    X = s(s(s(s(s(s(s(s(s(0)))))))))) ;
%    No
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

potencia(_X,0,s(0)).
potencia(X,s(Y),Z) :-
   potencia(X,Y,Z1),
   producto(X,Z1,Z).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 2.8 Definir la relación divisor_propio(X,Y) que se verifique
% si X es un divisor propio del número natural Y; es decir, X es un
% divisor de Y mayor 1 y menor que Y. Por ejemplo,
%    ?- divisor_propio(X,s(s(s(s(s(s(s(0))))))).
%    X = s(s(0)) ;
%    X = s(s(s(0))) ;
%    No
%    ?- divisor_propio(X,s(s(s(0)))).
%    No
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

divisor_propio(X,Y) :-
   menor(X,Y),
   menor(Z,Y),
   producto(Z,X,Y).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 2.9 Definir la relación compuesto(X) que se verifique si X
% es un número compuesto; es decir, que pose un divisor mayor que 1 y
```

```
% menor que X. Por ejemplo,
%    ?- compuesto(s(s(s(s(0))))).
%    Yes
%    ?- compuesto(s(s(s(0)))).
%    No
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

compuesto(X) :-
   divisor_propio(_Y,X).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 2.10 Definir la relación número_natural(X) que se verifique
% si X es un número natural. Por ejemplo,
%    ?- número_natural(s(s(0))).
%    Yes
%    ?- número_natural(s(s(3))).
%    No
%    ?- número_natural(X).
%    X = 0 ;
%    X = s(0) ;
%    X = s(s(0)) ;
%    X = s(s(s(0)))
%    Yes
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

número_natural(0).
número_natural(s(X)) :-
   número_natural(X).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 2.11 Definir la relación número_compuesto(X) que se
% verifique si X es un número compuesto; es decir, que posee un divisor
% mayor que 1 y menor que X. Por ejemplo,
%    ?- número_compuesto(X).
%    X = s(s(s(s(0)))) ;
%    X = s(s(s(s(s(s(0))))))
%    Yes
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

número_compuesto(X) :-
```

```
   número_natural(X),
   compuesto(X).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 2.12 Definir la relación entre(X,Y,Z) que se verifique si Z
% es un número natural mayor o igual que X y menor o igual que Y. Por
% ejemplo,
%    ?- entre(s(0),s(s(s(0))),X).
%    X = s(0) ;
%    X = s(s(0)) ;
%    X = s(s(s(0))) ;
%    No
%    ?- entre(s(s(s(0))),s(0),X).
%    No
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

entre(X,Y,X) :-
   menor(X,Y).
entre(X,X,X).
entre(X,Y,Z) :-
   menor(X,Y),
   entre(s(X),Y,Z).
```

7.2. Práctica 3a en Haskell

```
-- ---------------------------------------------------------------------
-- Importación de librerías auxiliares
-- ---------------------------------------------------------------------

import Test.QuickCheck
import Random

-- ---------------------------------------------------------------------
-- Ejercicio 1 (Relaciones familiares)
-- ---------------------------------------------------------------------
-- En este ejercicio se considera la siguiente base de conocimiento que
-- describe algunas relaciones familiares mediante la relación
-- (padre x y), que se verifica si x es padre de y, y (madre x y), que
-- se verifica si x es madre de y.
type Persona = String
```

```
personas :: [Persona]
personas = ["Ana", "Andrés", "Baltasar", "Belén", "Bernardo", "Carlos",
            "Carmen", "David", "Emilio", "Francisco"]

padre, madre :: Persona -> Persona -> Bool
padre "Andrés"   "Bernardo"  = True
padre "Andrés"   "Belén"     = True
padre "Andrés"   "Baltasar"  = True
padre "Baltasar" "David"     = True
padre "David"    "Emilio"    = True
padre "Emilio"   "Francisco" = True
padre _          _           = False
madre "Ana"      "Bernardo"  = True
madre "Ana"      "Belén"     = True
madre "Ana"      "Baltasar"  = True
madre "Belén"    "Carlos"    = True
madre "Belén"    "Carmen"    = True
madre _          _           = False
-- Se pide:
-- (a) Extender la base de conocimiento anterior mediante reglas para
--     definir las siguientes relaciones familiares:
--     (a.1) (progenitor x y) que se verifica si x es progenitor de y (es
--           decir, x es el padre o la madre de y),
--     (a.2) (abuelo x ,y) que se verifica si x es abuelo de y.
-- (b) Usar las relaciones definidas para responder a las siguientes
--     cuestiones:
--     (b.1) ¿quiénes son los progenitores de Bernardo?
--     (b.2) ¿quién es el abuelo de Carlos?
--     (b.3) ¿quiénes son los hijos de Andrés?
--     (b.4) ¿quiénes son los hijos de algún hijo de Ana?
-- ---------------------------------------------------------------------

-- (a.1) Definición de (progenitor x y)
progenitor :: Persona -> Persona -> Bool
progenitor x y = (padre x y) || (madre x y)

-- (a.2) Definición de (abuelo x y)
abuelo :: Persona -> Persona -> Bool
abuelo x y = or [(padre x z) && (progenitor z y) | z <- personas]
```

```
-- (b.1) ¿quiénes son los progenitores de Bernardo?
--    Main> [x | x <- personas, progenitor x "Bernardo"]
--    ["Ana","Andrés"]

-- (b.2) ¿quién es el abuelo de Carlos?
--    head [x | x <- personas, abuelo x "Carlos"]
--    "Andrés"

-- (b.3) ¿quiénes son los hijos de Andrés?
--    Main> [x | x <- personas, padre "Andrés" x]
--    ["Baltasar","Belén","Bernardo"]

-- (b.4) ¿quiénes son los hijos de algún hijo de Ana?
--    Main> [y | x <- personas, y <- personas,
--                madre "Ana" x, progenitor x y]
--    ["David","Carlos","Carmen"]

-- ---------------------------------------------------------------------
-- Ejercicio 2 (Números naturales)
-- ---------------------------------------------------------------------
-- En este ejercicio se usa la representación de los números naturales
-- construidos con 0 y s; es decir, la serie
--    0, S 0, S (S 0), S (S (S 0)), ...
-- representa a la serie de los números naturales
--    0, 1, 2, 3, ...
-- ---------------------------------------------------------------------
-- Ejercicio 2.1 Definir el tipo de datos Nat de los números naturales a
-- partir de la constante 0 (para el número 0) y la función S (para el
-- sucesor). Hacer Nat subclase de Eq y Show.
-- ---------------------------------------------------------------------

data Nat = 0 | S Nat
           deriving (Eq, Show)

-- ---------------------------------------------------------------------
-- Ejercicio 2.2 Definir la constante
--    nat :: [Nat]
-- cuyos elementos son los números naturales. Por ejemplo,
--    Main> take 5 nat
```

```
--    [O,S O,S (S O),S (S (S O)),S (S (S (S O)))]
-- ---------------------------------------------------------------------

nat :: [Nat]
nat = O:[S x | x <- nat]

-- ---------------------------------------------------------------------
-- Ejercicio 2.3 Definir la función
--    suma :: Nat -> Nat -> Nat
-- tal que (suma x y) es la suma de los números naturales x e y. Por
-- ejemplo,
--    Main> suma (S (S O)) (S (S (S O)))
--    S (S (S (S (S O))))
-- ---------------------------------------------------------------------

suma :: Nat -> Nat -> Nat
suma O     y = y
suma (S x) y = S (suma x y)

-- ---------------------------------------------------------------------
-- Ejercicio 2.4 Definir la relación
--    menor :: Nat -> Nat -> Bool
-- tal que (menor x y) se verifica si el número natural x es menor que
-- el número natural y. Por ejemplo,
--    Main> menor (S O) (S(S(O)))
--    True
--    Main> menor (S(S(O))) (S O)
--    False
--    Main> menor (S O) (S O)
--    False
-- ---------------------------------------------------------------------

menor :: Nat -> Nat -> Bool
menor O     (S _) = True
menor (S x) (S y) = menor x y
menor _     _     = False

-- ---------------------------------------------------------------------
-- Ejercicio 2.5 Definir la relación
--    mayor :: Nat -> Nat -> Bool
```

```
-- tal que (mayor x y) se verifica si el número natural x es mayor que
-- el número natural y. Por ejemplo,
--    Main> mayor (S O) (S(S(O)))
--    False
--    Main> mayor (S(S(O))) (S O)
--    True
--    Main> mayor (S O) (S O)
--    False
-- ---------------------------------------------------------------------

mayor :: Nat -> Nat -> Bool
mayor x y = menor y x

-- ---------------------------------------------------------------------
-- Ejercicio 2.6 Definir la función
--    resta :: Nat -> Nat -> Nat
-- tal que (resta x y) es x menos y si x es mayor igual que y y 0, en
-- caso contrario. Por ejemplo,
--    Main> resta (S(S(S(O)))) (S(O))
--    S (S O)
--    Main> resta (S(S(S(O)))) (S(S(O)))
--    S O
--    Main> resta (S(S(S(O)))) (S(S(S(O))))
--    O
--    Main> resta (S(S(S(O)))) (S(S(S(S(O)))))
--    O
-- ---------------------------------------------------------------------

resta :: Nat -> Nat -> Nat
resta x y
    | mayor x y = head [z | z <- nat, suma y z == x]
    | otherwise = O

-- ---------------------------------------------------------------------
-- Ejercicio 2.7 Definir la función
--    producto :: Nat -> Nat -> Nat
-- tal que (producto x y) es el producto de los números naturales x e
-- y. Por ejemplo,
--    Main> producto (S(S(O))) (S(S(S(O))))
--    S (S (S (S (S O))))
```

```
-- ---------------------------------------------------------------------

producto :: Nat -> Nat -> Nat
producto O     _ = O
producto (S x) y = suma y (producto x y)

-- ---------------------------------------------------------------------
-- Ejercicio 2.8 Definir la función
--    cuadrado :: Nat -> Nat
-- tal que (cuadrado x) es el cuadrado de x. Por ejemplo,
--    Main> cuadrado (S(S(O)))
--    S (S (S (S O)))
-- ---------------------------------------------------------------------

cuadrado :: Nat -> Nat
cuadrado x = producto x x

-- ---------------------------------------------------------------------
-- Ejercicio 2.9 Definir la función
--    cubo :: Nat -> Nat
-- tal que (cubo x) es el cubo de x. Por ejemplo,
--    Main> cubo (S(S(O)))
--    S (S (S (S (S (S (S O)))))))
-- ---------------------------------------------------------------------

cubo :: Nat -> Nat
cubo x = producto x (cuadrado x)

-- ---------------------------------------------------------------------
-- Ejercicio 2.10 Definir la función
--    potencia :: Nat -> Nat -> Nat
-- tal que (potencia x y) es la potencia y-ésima de x. Por ejemplo,
--    Main> potencia (S(S(O))) (S(S(S(O))))
--    S (S (S (S (S (S (S (S O)))))))
--    Main> potencia (S(S(S(O)))) (S(S(O)))
--    S (S (S (S (S (S (S (S O)))))))
-- ---------------------------------------------------------------------

potencia :: Nat -> Nat -> Nat
potencia _ O     = S O
```

```
potencia x (S y) = producto x (potencia x y)

-- ---------------------------------------------------------------------
-- Ejercicio 2.11 Definir la función
--    entre :: Nat -> Nat -> [Nat]
-- tal que (entre x y) es la lista de los números mayores o iguales que
-- x y menores o iguales que y. Por ejemplo,
--    Main> entre (S(O)) (S(S(S(O))))
--    [S O,S (S O),S (S (S O))]
--    Main> entre (S(S(S(O)))) (S(O))
--    []
-- ---------------------------------------------------------------------

entre :: Nat -> Nat -> [Nat]
entre x y
    | mayor x y = []
    | otherwise = x:(entre (S x) y)

-- ---------------------------------------------------------------------
-- Ejercicio 2.12 Definir la relación
--    divisor_propio :: Nat -> Nat -> Bool
-- tal que (divisor_propio x y) se verifica si x es un divisor propio
-- del número natural y; es decir, x es un divisor de y mayor 1 y menor
-- que y. Por ejemplo,
--    Main> divisor_propio (S(S(O))) (S(S(S(S(O)))))
--    True
--    Main> divisor_propio (S(S(O))) (S(S(S(S(S(O))))))
--    False
-- ---------------------------------------------------------------------

divisor_propio :: Nat -> Nat -> Bool
divisor_propio x y =
    menor x y &&
    or [producto x z == y | z <- entre (S(S(O))) y]

-- ---------------------------------------------------------------------
-- Ejercicio 2.13 Definir la relación
--    compuesto :: Nat -> Bool
-- tal que (compuesto x) que se verifica si x es un número compuesto; es
-- decir, que posee un divisor mayor que 1 y menor que x. Por ejemplo,
```

```
--    Main> compuesto (S(S(S(S(O)))))
--    True
--    Main> compuesto (S(S(S(O))))
--    False
-- ----------------------------------------------------------------------

compuesto :: Nat -> Bool
compuesto x =
    or [divisor_propio y x | y <- (entre (S(S(O))) x)]

-- ----------------------------------------------------------------------
-- Ejercicio 2.14 Definir la constantes
--    números_compuestos :: [Nat]
-- cuyo valor es la lista de los números compuestos. Por ejemplo,
--    Main> take 2 números_compuestos
--    [S (S (S (S O))),S (S (S (S (S O)))))]
-- ----------------------------------------------------------------------

números_compuestos :: [Nat]
números_compuestos =
    [x | x <- nat, compuesto x]


-- ----------------------------------------------------------------------

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

natural :: Gen Nat
natural =
    oneof
    [ return O
    , do n <- natural
         return (S n)
    ]
```

```
instance Arbitrary Nat where
    arbitrary = natural

prop_SumaConmutativa :: Nat -> Nat -> Bool
prop_SumaConmutativa x y =
    suma x y == suma y x

prop_SumaAsociativa  :: Nat -> Nat -> Nat -> Bool
prop_SumaAsociativa x y z =
    suma x (suma y z) == suma (suma x y) z
```

Capítulo 8

Práctica 3b (10 de diciembre de 2007)

8.1. Práctica 3b en Prolog

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 1 Definir, usando append, la  relación pertenece(X,L) que se
% verifique si X es un elemento de la lista L. Por ejemplo,
%    ?- pertenece(b,[a,b]).
%    Yes
%    ?- pertenece(c,[a,b]).
%    No
%    ?- pertenece(X,[a,b]).
%    X = a ;
%    X = b ;
%    No
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

pertenece(X,L) :-
   append(_,[X|_],L).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 2 Definir la relación sublista(L1,L2) que se verifique si L1
% es una sublista de L2. Por ejemplo,
%   ?- sublista([b,c],[a,b,c,d]).
%   Yes
%   ?- sublista([b,d],[a,b,c,d]).
%   No
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

sublista(L1,L2) :-
```

```
   append(_,L4,L2),
   append(L1,_,L4).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 3 Definir la relación último(X,L) que se verifique si X es
% el último elemento de la lista L. Por ejemplo,
%     ?- último(X,[a,b,c]).
%     X = c ;
%     No
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Definición usando append:
último_1(X,L) :-
   append(_,[X],L).

% Definición recursiva:
último_2(X,[X]).
último_2(X,[_|L]) :-
   último_2(X,L).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 4 Definir la relación inversa(L1,L2) que se verifique si L2
% es la lista L1 en orden inverso, Por ejemplo,
%   ?- inversa([a,b,c],L).
%   L = [c, b, a] ;
%   No
%   ?- inversa([a,[b,c],d,e],L).
%   L = [e, d, [b, c], a] ;
%   No
%  Nota: La relación predefinida correspondiente a inversa es reverse.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

inversa([],[]).
inversa([X|L1],L2) :-
   inversa(L1,L3),
   append(L3,[X],L2).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 5 Definir la relación palíndromo(L) que se verifique si L es
% un palíndromo; es decir, da igual leerlo de izquierda a derecha que
```

```
% leerlo de derecha a izquierda. Por ejemplo,
%    ?- palíndromo([r,o,m,a,y,a,m,o,r]).
%    Yes
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

palíndromo(L) :-
   inversa(L,L).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 6 Definir la relación selecciona(X,L1,L2) que se verifique
% si X es un elemento de la lista L1 y L2 es la lista de los restantes
% elementos. Por ejemplo,
%   ?- selecciona(X,[a,b,c],L).
%   X = a     L = [b, c] ;
%   X = b     L = [a, c] ;
%   X = c     L = [a, b] ;
%   No
% Nota: La relación predefinida correspondiente a selecciona es select.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

selecciona(X,[X|L],L).
selecciona(X,[Y|L1],[Y|L2]) :-
   selecciona(X,L1,L2).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 7 Definir la relación máximo(X,Y,Z) que se verifique si Z es
% el máximo de los números X e Y. Por ejemplo,
%    ?- máximo(3,5,Z).
%    Z=5
%    ?- máximo(2,3,X).
%    X = 3
%    ?- máximo(3,2,X).
%    X = 3
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Primera solución:
máximo(X,Y,X) :-
   X >= Y.
máximo(X,Y,Y) :-
   X < Y.
```

```
% Segunda solución, usando max es
máximo_2(X,Y,Z) :-
   Z is max(X,Y).

% Hay que tener en cuenta la diferencia de comportamiento si no están
% instanciados los dos primeros argumentos.
%   ?- máximo(3,5,Z).
%   Z = 5
%   ?- máximo(3,Y,5).
%   Y = 5
%   ?- máximo_2(3,5,Z).
%   Z = 5
%   ?- máximo_2(3,Y,5).
%   ERROR: is/2: Arguments are not sufficiently instantiated

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 8 (Algoritmo de Euclides) Dados dos enteros positivos X e Y,
% el máximo común divisor (mcd) D puede obtenerse de la siguiente
% manera:
%  * Si X e Y son iguales, entonces D es igual a X
%  * Si X<Y, entonces D es igual al máximo común divisor de X y la
%    diferencia Y-X.
%  * Si Y<X entonces hacemos lo mismo que en caso anterior con X e Y
%    intercambiados.
% Definir la relación mcd(X,Y,D) que calcule el mcd D de los enteros
% positivos X e Y.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

mcd(X,X,X).
mcd(X,Y,Z) :-
   X < Y,
   Y1 is Y - X,
   mcd(X,Y1,Z).
mcd(X,Y,Z) :-
   X > Y,
   mcd(Y,X,Z).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 9 Definir la relación longitud(L,N) que se verifique si N es
```

```
% la longitud de la lista L. Por ejemplo,
%    ?- longitud([],N).
%    N = 0
%    ?- longitud([a,b,c],N).
%    N = 3
%    ?- longitud([a,[b,c]],N).
%    N = 2
% Nota: La relación predefinida correspondiente a longitud es length.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

longitud([],0).
longitud([_X|L],N) :-
   longitud(L,M),
   N is M + 1.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 10 Definir la relación máximo_lista(L,N) que se verifique si
% N es el máximo de los elementos de la lista de números L. Por ejemplo,
%    ?- máximo_lista([1,3,9,5],N).
%    N = 9
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Primera solución:
máximo_lista([X],X).
máximo_lista([X,Y|L],N) :-
   máximo(X,Y,Z),
   máximo_lista([Z|L],N).

% Segunda solución, usando max:
máximo_lista_2([X],X).
máximo_lista_2([X,Y|L],N) :-
   Z is max(X,Y),
   máximo_lista_2([Z|L],N).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 11 Definir la relación entre(N1,N2,X) que se verifique si X
% es mayor o igual que N1 y menor o igual que N2. Por ejemplo,
%    ?- entre(2,5,X).
%    X = 2 ;
%    X = 3 ;
```

```
%    X = 4 ;
%    X = 5 ;
%    No
%    ?- entre(2,1,X).
%    No
% Nota: La relación predefinida correspondiente a entre es between.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

entre(N1,N2,N1) :-
   N1 =< N2.
entre(N1,N2,X) :-
   N1 < N2,
   N3 is N1 + 1,
   entre(N3,N2,X).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 12 (Base de datos familiar)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% En este ejercicio vamos a considerar el programa familia.pl del Tema
% 8. Las personas se representan mediante estructuras de la
% forma
persona([Nombre,Apellido_1,Apellido_2],
        fecha(Día,Mes,Año),
        trabajo(Profesión,Sueldo))
% La base de datos de las familias se definen por la relación
% familia(P,M,L) que se verifica si existe una familia con padre P,
% madre M y lista de hijos L.
familia(persona([tomas,garcia,perez],
                fecha(7,mayo,1950),
                trabajo(profesor,75)),
        persona([ana,lopez,ruiz],
                fecha(10,marzo,1952),
                trabajo(medica,100)),
        [ persona([juan,garcia,lopez],
                  fecha(5,enero,1970),
                  estudiante),
          persona([maria,garcia,lopez],
                  fecha(12,abril,1972),
                  estudiante) ]).
familia(persona([jose,perez,ruiz],
```

```
                fecha(6,marzo,1953),
                trabajo(pintor,150)),
        persona([luisa,galvez,perez],
                fecha(12,mayo,1954),
                trabajo(medica,100)),
        [ persona([juan_luis,perez,perez],
                  fecha(5,febrero,1980),
                  estudiante),
          persona([maria_jose,perez,perez],
                  fecha(12,junio,1982),
                  estudiante),
          persona([jose_maria,perez,perez],
                  fecha(12,julio,1984),
                  estudiante) ]).
% La relación casado(X) que se verifica si X es un hombre casado y la
% relación casada(X) que se verifica si X es un mujer casada.
casado(X) :-
   familia(X,_,_).

casada(X) :-
   familia(_,X,_).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 12.1 Definir la relación hijo(X) que se verifique si X figura
% en alguna lista de hijos.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

hijo(X) :-
   familia(_,_,L),
   member(X,L).

% Los hijos se obtienen con la siguiente consulta
%    ?- hijo(X).
%    X = persona([juan,garcía,lópez],fecha(5,enero,1970),estudiante) ;
%    X = persona([maría,garcía,lópez],fecha(12,abril,1972),estudiante) ;
%    X = persona([juan_luis,pérez,pérez],fecha(5,febrero,1980),estudiante) ;
%    X = persona([maría_josé,pérez,pérez],fecha(12,junio,1982),estudiante) ;
%    X = persona([josé_maría,pérez,pérez],fecha(12,julio,1984),estudiante) ;
%    No
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 12.2 Definir la relación existe(X) que se verifique si X es
% una persona existente en la base de datos.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

existe(X) :-
   casado(X);
   casada(X);
   hijo(X).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 12.3 Hacer una pregunta tal que que las respuestas sean las
% listas de la forma [nombre, apellido1, apellido2] de todas las
% personas que existen.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% La consulta sobre todas las personas es
%    ?- existe(persona(X,_,_)).
%    X = [tomás,garcía,pérez] ;
%    X = [josé,pérez,ruiz] ;
%    X = [ana,lópez,ruiz] ;
%    X = [luisa,gálvez,pérez] ;
%    X = [juan,garcía,lópez] ;
%    X = [maría,garcía,lópez] ;
%    X = [juan_luis,pérez,pérez] ;
%    X = [maría_josé,pérez,pérez] ;
%    X = [josé_maría,pérez,pérez] ;
%    No

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 12.4 Determinar todos los estudiantes nacidos antes de 1983.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% La consulta sobre los estudiantes nacidos antes de 1983 es
%    ?- existe(persona(X,fecha(_,_,Año),estudiante)),
%       Año < 1983.
%    X = [juan,garcía,lópez]        Año = 1970 ;
%    X = [maría,garcía,lópez]       Año = 1972 ;
%    X = [juan_luis,pérez,pérez]    Año = 1980 ;
```

```
%    X = [maría_josé,pérez,pérez]   Año = 1982 ;
%    No

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 12.5 Definir la relación fecha_de_nacimiento(X,Y) de forma
% que si X es una persona, entonces Y es su fecha de nacimiento.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

fecha_de_nacimiento(persona(_,Y,_),Y).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 12.6 Buscar todos los hijos nacidos en 1982.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% La consulta sobre los hijos nacidos en 1982 es
%    ?- hijo(X), fecha_de_nacimiento(X,fecha(_,_,1982)).
%    X = persona([maría_josé,pérez,pérez], fecha(12,junio,1982), estudiante) ;
%    No

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 12.7 Definir la relación sueldo(X,Y) que se verifique si el
% sueldo de la persona X es Y.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

sueldo(persona(_,_,trabajo(_,Y)),Y).
sueldo(persona(_,_,estudiante),0).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 12.8 Buscar todas las personas nacidas antes de 1954 cuyo
% sueldo sea superior a 72 euros.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% La consulta sobre las personas nacidas antes de 1954 cuyo sueldo sea
% superior a 12.000 es
%    ?- existe(X),
%       fecha_de_nacimiento(X,fecha(_,_,Año)),
%       Año < 1954,
%       sueldo(X,Y),
%       Y > 72.
%    X = persona([tomás,garcía,pérez],fecha(7,mayo,1950),trabajo(profesor,75))
```

```
%    Año = 1950
%    Y = 75 ;
%    X = persona([josé,pérez,ruiz],fecha(6,marzo,1953),trabajo(pintor,150))
%    Año = 1953
%    Y = 150 ;
%    X = persona([ana,lópez,ruiz],fecha(10,marzo,1952),trabajo(medica,100))
%    Año = 1952
%    Y = 100 ;
%    No

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 12.9 Definir la relación total(L,Y) que se verifique si Y es
% la suma de los sueldos de las personas de la lista L.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

total([],0).
total([X|L],Y) :-
   sueldo(X,Y1),
   total(L,Y2),
   Y is Y1 + Y2.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 12.10 Calcular los ingresos totales de cada familia.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% La consulta sobre los sueldos totales de cada familia es
%    ?- familia(X,Y,Z),
%       total([X,Y|Z],Total).
%    X = persona([tomás,garcía,pérez],fecha(7,mayo,1950),trabajo(profesor,75))
%    Y = persona([ana,lópez,ruiz],fecha(10,marzo,1952),trabajo(medica,100))
%    Z = [persona([juan,garcía,lópez],fecha(5,enero,1970),estudiante),
%         persona([maría,garcía,lópez],fecha(12,abril,1972),estudiante)]
%    Total = 175 ;
%    X = persona([josé,pérez,ruiz],fecha(6,marzo,1953),trabajo(pintor,150))
%    Y = persona([luisa,gálvez,pérez],fecha(12,mayo,1954),trabajo(medica,100))
%    Z = [persona([juan_luis,pérez,pérez],fecha(5,febrero,1980),estudiante),
%         persona([maría_josé,pérez,pérez],fecha(12,junio,1982),estudiante),
%         persona([josé_maría,pérez,pérez],fecha(12,julio,1984),estudiante)]
%    Total = 250 ;
%    No
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 13 (Simulación del autómata)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% En este ejercicio consideraremos el programa automata.pl del Tema
% 8. El autómata se define mediante las siguientes relaciones
%  * final(E) que se verifica si E es el estado final.
%  * trans(E1,X,E2) que se verifica si se puede pasar del estado E1 al
%    estado E2 usando la letra X.
%  * nulo(E1,E2) que se verifica si se puede pasar del estado E1 al
%    estado E2 mediante un movimiento nulo.
% El autómata del ejemplo está definido por
final(e3).

trans(e1,a,e1).
trans(e1,a,e2).
trans(e1,b,e1).
trans(e2,b,e3).
trans(e3,b,e4).

nulo(e2,e4).
nulo(e3,e1).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 13.1 Definir la relación acepta_acotada(S,L,N) que se
% verifique si el autómata, a partir del estado S, acepta la lista L y
% la longitud de L es N.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Primera definición:
acepta_acotada_a(E,L,N) :-
   length(L,N),
   acepta_acotada_a_aux(E,L).

% acepta_acotada_a_aux(E,L) se verifica si el autómata, a partir del
% estado E acepta la lista L
acepta_acotada_a_aux(E,[]) :-
   final(E).
acepta_acotada_a_aux(E,[X|L]) :-
```

```
   trans(E,X,E1),
   acepta_acotada_a_aux(E1,L).
acepta_acotada_a_aux(E,L) :-
   nulo(E,E1),
   acepta_acotada_a_aux(E1,L).

% Segunda definición:
acepta_acotada_b(E,[],0) :-
   final(E).
acepta_acotada_b(E,[X|L],N) :-
   N > 0,
   trans(E,X,E1),
   M is N -1,
   acepta_acotada_b(E1,L,M).
acepta_acotada_b(E,L,N) :-
   nulo(E,E1),
   acepta_acotada_b(E1,L,N).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 13.2  Buscar las cadenas aceptadas a partir de e1 con
% longitud 3.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% La consulta es
%    ?- acepta_acotada_a(e1,L,3).
%    L = [a, a, b] ;
%    L = [b, a, b] ;
%    No
%    ?- acepta_acotada_b(e1,L,3).
%    L = [a, a, b] ;
%    L = [b, a, b] ;
%    No

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 13.3 Definir la relación acepta_acotada_2(S,L,N) que se
% verifique si el autómata, a partir del estado S, acepta la lista L y
% la longitud de L es menor o igual que N.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Primera definición
```

```
acepta_acotada_2_a(E,L,N) :-
   between(0,N,M),
   length(L,M),
   acepta_acotada_a_aux(E,L).

% Segunda definición:
acepta_acotada_2_b(E,[],_N) :-
   final(E).
acepta_acotada_2_b(E,[X|L],N) :-
   N > 0,
   trans(E,X,E1),
   M is N-1,
   acepta_acotada_2_b(E1,L,M).
acepta_acotada_2_b(E,L,N) :-
   nulo(E,E1),
   acepta_acotada_2_b(E1,L,N).

% La consulta sobre las cadenas aceptadas a partir de e1 con longitud
% menor o igual a 3 es
%    ?- acepta_acotada_2_a(e1,L,3).
%    L = [a, b] ;
%    L = [a, a, b] ;
%    L = [b, a, b] ;
%    No
%    ?- acepta_acotada_2_b(e1,L,3).
%    L = [a, a, b] ;
%    L = [a, b] ;
%    L = [b, a, b] ;
%    No

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 14 (Banda de músicos)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Se sabe que
%  1. Una banda está compuesta por tres músicos de distintos paises y
%     que tocan distintos instrumentos.
%  2. El pianista toca primero.
%  3. Juan toca el saxo y toca antes que el australiano.
%  4. Marcos es francés y toca antes que el violinista.
%  5. Hay un músico japonés.
```

```
%  6. Un músico se llama Saúl.
%  Determinar el nombre, el país y el instrumento que toca cada uno de
%  los músicos de la banda.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% La relación solución_músicos(S) se verifica si S es una solución del
% problema de los músicos.
solución_músicos(S) :-
   % 1. Una banda está compuesta por tres músicos de distintos paises
   %    y que tocan distintos instrumentos.
   presolución(S),
   % 2. El pianista toca primero.
   instrumento(X,piano),
   primero(X,S),
   % 3. Juan toca el saxo y toca antes que el australiano.
   nombre(Y,juan),
   instrumento(Y,saxo),
   país(Z,australia),
   antes(Y,Z,S),
   % 4. Marcos es francés y toca antes que el violinista.
   nombre(Y1,marco),
   país(Y1,francia),
   instrumento(Z1,violín),
   antes(Y1,Z1,S),
   % 5. Hay un músico japonés.
   pertenece(U,S),
   país(U,japón),
   % 6. Un músico se llama Saúl.
   pertenece(V,S),
   nombre(V,saúl).

% La definición se basa en relación presolución(S) que se verifica si S
% es una presolución; i.e. S es un término de la forma
%    banda(músico(N1,P1,I1),
%          músico(N2,P2,I2),
%          músico(N3,P3,I3))
% donde músico(N,P,I) representa al músico de nombre N, país P y que
% toca el instrumento I.
presolución(banda(músico(_N1,_P1,_I1),
                  músico(_N2,_P2,_I2),
```

```
                músico(_N3,_P3,_I3))).

% Además, se usan las siguientes relaciones auxiliares:
% nombre(X,N)  se verifica si N es el el nombre del músico X.
nombre(músico(N,_,_),N).

% país(X,P) se verifica si P es el país del músico X.
país(músico(_,P,_),P).

% instrumento(X,I) se verifica si I es el instrumento que toca  el
% músico X.
instrumento(músico(_,_,I),I).

% primero(X,B) se verifica si X es el primer músico de la banda  B.
primero(X,banda(X,_,_)).

% pertenece(X,B) se verifica si X es un músico de la banda B.
pertenece(X,banda(X,_,_)).
pertenece(X,banda(_,X,_)).
pertenece(X,banda(_,_,X)).

% antes(X,Y,B) se verifica si X toca antes que Y en la banda  B.
antes(X,Y,banda(X,Y,_)).
antes(X,Y,banda(X,_,Y)).
antes(X,Y,banda(_,X,Y)).

% La consulta para obtener la solución es
%    ?- solución_músicos(S).
%    S = [músico(marco, francia,   piano),
%         músico(juan,  japón,     saxo),
%         músico(saúl,  australia, violín)] ;
%    No

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 15 (Movimientos del caballo)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Supongamos que los cuadros del tablero de ajedrez los representamos
% por pares de números [X,Y] con X e Y entre 1 y 8.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% Ejercicio 15.1 Definir la relación salta(+C1,?C2) que se verifique si
% el caballo puede pasar en un movimiento del cuadrado C1 al cuadrado
% C2. Por ejemplo,
%     ?- salta([1,1],S).
%     S=[3,2];
%     S=[2,3];
%     No
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

salta([X,Y],[X1,Y1]) :-
   dxy(Dx,Dy),
   X1 is X+Dx,
   correcto(X1),
   Y1 is Y+Dy,
   correcto(Y1).

% dxy(?X,?Y) se verifica si un caballo puede moverse X espacios
% horizontales e Y verticales.
dxy(2,1).
dxy(2,-1).
dxy(-2,1).
dxy(-2,-1).
dxy(1,2).
dxy(1,-2).
dxy(-1,2).
dxy(-1,-2).

% correcto(+X) se verifica si X está entre 1 y 8.
correcto(X) :-
   1 =< X,
   X =< 8.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 15.2 Definir la relación camino(L) que se verifique si L es
% una lista de cuadrados representando el camino recorrido por un
% caballo sobre un tablero vacío. Por ejemplo,
%     ?- camino([[1,1],C]).
%     C=[3,2];
%     C=[2,3];
%     No
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

camino([_]).
camino([C1,C2|L]) :-
   salta(C1,C2),
   camino([C2|L]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 15.3 Usando la relación camino, escribir una pregunta para
% determinar los caminos de longitud 4 por los que puede desplazarse un
% caballo desde cuadro [2,1] hasta el otro extremo del tablero (Y=8) de
% forma que en el segundo movimiento pase por el cuadro [5,4].
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%   ?- camino([[2,1],C1,[5,4],C2,[X,8]]).
%   C1 = [4, 2]   C2 = [6, 6]   X = 7 ;
%   C1 = [4, 2]   C2 = [6, 6]   X = 5 ;
%   C1 = [4, 2]   C2 = [4, 6]   X = 5 ;
%   C1 = [4, 2]   C2 = [4, 6]   X = 3 ;
%   C1 = [3, 3]   C2 = [6, 6]   X = 7 ;
%   C1 = [3, 3]   C2 = [6, 6]   X = 5 ;
%   C1 = [3, 3]   C2 = [4, 6]   X = 5 ;
%   C1 = [3, 3]   C2 = [4, 6]   X = 3 ;
%   No

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 15.4 Calcular el menor número de movimientos necesarios para
% desplazar el caballo del cuadro [1,1] al [2,2]. ¿Cuántos caminos de
% dicha longitud hay de [1,1] a [2,2]?
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%   ?- camino([[1,1],_,[2,2]]).
%   No
%   ?- camino([[1,1],_,_,[2,2]]).
%   No
%   ?- camino([[1,1],_,_,_,[2,2]]).
%   Yes
%   ?- camino([[1,1],C2,C3,C4,[2,2]]).
%   C2 = [3, 2]   C3 = [5, 3]   C4 = [3, 4] ;
%   C2 = [3, 2]   C3 = [5, 3]   C4 = [4, 1] ;
```

```
%   C2 = [3, 2]   C3 = [5, 1]   C4 = [4, 3] ;
%   C2 = [3, 2]   C3 = [1, 3]   C4 = [3, 4] ;
%   C2 = [3, 2]   C3 = [2, 4]   C4 = [4, 3] ;
%   C2 = [2, 3]   C3 = [4, 2]   C4 = [3, 4] ;
%   C2 = [2, 3]   C3 = [3, 5]   C4 = [1, 4] ;
%   C2 = [2, 3]   C3 = [3, 5]   C4 = [4, 3] ;
%   C2 = [2, 3]   C3 = [3, 1]   C4 = [4, 3] ;
%   C2 = [2, 3]   C3 = [1, 5]   C4 = [3, 4] ;
%   No
```

8.2. Práctica 3b en Haskell

```
-- ---------------------------------------------------------------------
-- Importación de librerías auxiliares
-- ---------------------------------------------------------------------

import Test.QuickCheck
import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1 Definir la relación
--    pertenece :: Eq a => a -> [a] -> Bool
-- tal que (pertenece x ys) que se verifique si x es un elemento de la
-- lista ys. Por ejemplo,
--    pertenece 'b' "ab"  ==>  True
--    pertenece 'c' "ab"  ==>  False
-- ---------------------------------------------------------------------

pertenece :: Eq a => a -> [a] -> Bool
pertenece x ys = elem x ys

-- ---------------------------------------------------------------------
-- Ejercicio 2 Definir la relación
--    sublista :: Eq [a] => [a] -> [a] -> Bool
-- tal que (sublista xs ys) se verifique si xs es una sublista de ys. Por
-- ejemplo,
--    sublista "bc" "abcd"  ==>  True
--    sublista "bd" "abcd"  ==>  False
-- ---------------------------------------------------------------------
```

```
sublista :: Eq [a] => [a] -> [a] -> Bool
sublista xs ys =
    elem xs (sublistas ys)

-- (sublistas xs) es la lista de las sublistas de xs. Por ejemplo,
--    Main> sublistas "abcd"
--    ["a","ab","abc","abcd","b","bc","bcd","c","cd","d",""]
sublistas xs =
    concat [prefijosPropios ys | ys <- sufijosPropios xs] ++ [[]]

-- (prefijosPropios xs) es la lista de los prefijos de xs no vacío. Por
-- ejemplo,
--    prefijosPropios "abc"  ==>  ["a","ab","abc"]
prefijosPropios xs = [take n xs | n <- [1..length xs]]

-- (sufijosPropios xs) es la lista de los sufijos de xs no vacío. Por
-- ejemplo,
-- sufijosPropios "abc"  ==>  ["abc","bc","c"]
sufijosPropios xs = [drop n xs | n <- [0..(length xs)-1]]

-- Otra definición de sublista es
sublistas2 :: [a] -> [[a]]
sublistas2 xs =
    [take n (drop m xs) | m <- [0..len], n <- [1..len-m]] ++ [[]]
    where len = length xs

-- Las dos definiciones dan el mismo conjunto.
prop_sublistas :: [Int] -> Bool
prop_sublistas xs =
    sort(sublistas xs) == sort(sublistas2 xs)

-- En efecto,
--    Main> quickCheck prop_sublistas
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 3 Definir la función
--    ultimo :: [a] -> a
-- tal que (ultimo xs) es el último elemento de la lista xs. Por ejemplo,
```

```
--     ultimo "abc"  ==>  'c'
-- Nota: La función predefinida correspondiente a ultimo es last.
-- ---------------------------------------------------------------------

ultimo :: [a] -> a
ultimo [x]    = x
ultimo (x:xs) = ultimo xs

-- ---------------------------------------------------------------------
-- Ejercicio 4 Definir la función
--    inversa :: [a] -> [a]
-- tal que (inversa xs) es la lista xs en orden inverso, Por ejemplo,
--    inversa "abcd"  ==>  "dcba"
--  Nota: La relación predefinida correspondiente a inversa es reverse.
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 5 Definir la relación
--    palíndromo :: Eq [a] => [a] -> Bool
-- tal que (palíndromo xs) que se verifique si xs es un palíndromo; es
-- decir, da igual leerlo de izquierda a derecha que leerlo de derecha a
-- izquierda. Por ejemplo,
--    palíndromo "romayamor"  ==>  True
-- ---------------------------------------------------------------------

palíndromo :: Eq [a] => [a] -> Bool
palíndromo xs = inversa xs == xs

-- ---------------------------------------------------------------------
-- Ejercicio 6 Definir la función
--    selecciona :: [a] -> [(a,[a])]
-- tal que (selecciona xs) es la lista de pares formados por un elemento
-- de la lista xs y la lista de los restantes elementos. Por ejemplo,
--    selecciona "abc"  ==>  [('a',"bc"),('b',"ac"),('c',"ab")]
-- ---------------------------------------------------------------------
```

```
selecciona :: [a] -> [(a,[a])]
selecciona xs =
    [(xs !! n, (take n xs) ++ (drop (n+1) xs)) | n <- [0..(length xs)-1]]

-- ---------------------------------------------------------------------
-- Ejercicio 7 Definir la función
--    máximo :: Int -> Int -> Int
-- tal que (máximo x y) es el máximo de los números x e y. Por ejemplo,
--    máximo 2 5  ==>  5
--    máximo 5 2  ==>  5
-- ---------------------------------------------------------------------

-- Primera solución:
máximo :: Int -> Int -> Int
máximo x y
    | x >= y    = x
    | otherwise = y

-- Segunda solución, usando max es
máximo_2 :: Int -> Int -> Int
máximo_2 = max

-- ---------------------------------------------------------------------
-- Ejercicio 8 (Algoritmo de Euclides) Dados dos enteros positivos x e
-- y, el máximo común divisor (mcd) puede obtenerse de la siguiente
-- manera:
--  * Si x e y son iguales, entonces el mcd es igual a x
--  * Si x<y, entonces el mcd es igual al máximo común divisor de x y la
--    diferencia y-x.
--  * Si y<x entonces hacemos lo mismo que en caso anterior con x e y
--    intercambiados.
-- Definir la función
--    mcd :: Int -> Int -> Int
-- tal que (mcd x y) es el máximo común divisor de x e y.
-- ---------------------------------------------------------------------

mcd :: Int -> Int -> Int
mcd x y
    | x == y    = x
    | x <  y    = mcd x (y-x)
```

```
    | otherwise = mcd y x

-- ---------------------------------------------------------------------
-- Ejercicio 9 Definir la función
--    longitud :: [a] -> Int
-- tal que (logitud xs) es la longitud de la lista xs. Por ejemplo,
--    longitud "abc"  ==>  3
-- Nota: La función predefinida correspondiente a longitud es length.
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 10 Definir la función
--    máximo_lista :: Ord a => [a] -> a
-- tal que (máximo_lista xs) es el máximo de los elementos de la lista
-- xs. Por ejemplo,
--    máximo_lista [1,3,9,5]  ==>  9
-- Nota: La función predefinida correspondiente a máximo_lista es
-- maximum.
-- ---------------------------------------------------------------------

-- Primera solución:
máximo_lista :: Ord a => [a] -> a
máximo_lista [x] = x
máximo_lista (x:y:xs) = máximo_lista ((max x y):xs)

-- Segunda solución, usando max:
máximo_lista_2 :: Ord a => [a] -> a
máximo_lista_2 [x] = x
máximo_lista_2 (x:y:xs) = max x (máximo_lista_2 (y:xs))

-- ---------------------------------------------------------------------
-- Ejercicio 11 Definir la función
--    entre :: Int -> Int -> [Int]
-- tal que (entre x y) es la lista de números mayores o iguales que x y
-- menores o iguales que y. Por ejemplo,
--    entre 2 5  ==>  [2,3,4,5]
```

```
--    entre 2 1  ==>  []
-- ---------------------------------------------------------------------

-- Definición recursiva:
entre :: Int -> Int -> [Int]
entre x y
    | x > y = []
    | x == y = [x]
    | x < y  = x:(entre (x+1) y)

-- Definición con intervalos numéricos:
entre2 :: Int -> Int -> [Int]
entre2 x y = [x..y]

-- ---------------------------------------------------------------------
-- Ejercicio 12 (Representación de segmentos)
-- ---------------------------------------------------------------------
-- Ejercicio 12.1 Definir el tipo de datos Punto para representar los
-- puntos del plano como un par de números reales.
-- ---------------------------------------------------------------------

data Punto = P Float Float

-- ---------------------------------------------------------------------
-- Ejercicio 12.2 Definir el tipo de datos Segmento para representar los
-- segmentos como un par de puntos.
-- ---------------------------------------------------------------------

data Segmento = S Punto Punto

-- ---------------------------------------------------------------------
-- Ejercicio 12.3 Definir la relación
--    horizontal :: Segmento -> Bool
-- tal que (horizontal s) se verifica si el segmento s es
-- horizontal. Por ejemplo,
--    horizontal (S (P 1 2) (P 1 3))  ==>  True
--    horizontal (S (P 1 2) (P 4 2))  ==>  False
-- ---------------------------------------------------------------------

horizontal :: Segmento -> Bool
```

```
horizontal (S (P x _) (P y _)) = x == y

-- ---------------------------------------------------------------------
-- Ejercicio 12.4 Definir la relación
--    vertical :: Segmento -> Bool
-- tal que (vertical s) se verifica si el segmento s es
-- vertical. Por ejemplo,
--    vertical (S (P 1 2) (P 4 2))  ==>  True
--    vertical (S (P 1 2) (P 1 3))  ==>  False
-- ---------------------------------------------------------------------

vertical :: Segmento -> Bool
vertical (S (P _ x) (P _ y)) = x == y

-- ---------------------------------------------------------------------
-- Ejercicio 13 (Base de datos familiar)
-- ---------------------------------------------------------------------
-- En este ejercicio vamos a considerar ña versión Haskell del programa
-- familia.pl del Tema 8. Las personas se representan mediante
-- el tipo
type Persona          = (NombreApellidos, Fecha, Trabajo)
type NombreApellidos  = (Nombre, Apellido, Apellido)
type Nombre           = String
type Apellido         = String
type Fecha            = (Día, Mes, Año)
type Día              = Int
type Mes              = String
type Año              = Int
type Trabajo          = (Profesión, Sueldo)
type Profesión        = String
type Sueldo           = Int
-- Una familia está compuesta por el padre la madre y la lista de
-- hijos. El tipo para representar las familias es
type Familia = (Persona, Persona, [Persona])

-- La base de datos de las familias se definen por la constante familias
familias :: [Familia]
familias = [familia1, familia2]

familia1, familia2 :: Familia
```

```
familia1 =
    ((("Tomas", "Garcia","Perez"),( 7,"Mayo", 1950),("Profesor",   75)),
     (("Ana",   "Lopez", "Ruiz"), (10,"Marzo",1952),("Medica",    100)),
     [(("Juan", "Garcia","Lopez"),( 5,"Enero",1970),("Estudiante",  0)),
      (("Maria","Garcia","Lopez"),(12,"Abril",1972),("Estudiante",  0))])
familia2 =
    ((("Jose","Perez","Ruiz"),        (6,"Marzo",1953),    ("Pintor",150)),
     (("Luisa","Perez","Galvez"),     (12,"Mayo",1954),    ("Médica",100)),
     [(("Juan Luis","Perez","Perez"), ( 5,"Febrero",1980), ("Estudiante", 0)),
      (("Maria Jose","Perez","Perez"),(12,"Junio",1982),   ("Estudiante", 0)),
      (("Jose Maria","Perez","Perez"),(12,"Julio",1984),   ("Estudiante", 0))])
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 13.1 Definir la función
--    padreFamilia :: Familia -> Persona
-- tal que (padreFamilia f) es el padre de la familia f. Por ejemplo,
--    Main> padreFamilia familia1
--    (("Tomas","Garcia","Perez"),(7,"Mayo",1950),("Profesor",75))
-- ---------------------------------------------------------------------

padreFamilia :: Familia -> Persona
padreFamilia (p,_,_) = p

-- ---------------------------------------------------------------------
-- Ejercicio 13.2 Definir la función
--    madreFamilia :: Familia -> Persona
-- tal que (madreFamilia f) es la madre de la familia f. Por ejemplo,
--    Main> madreFamilia familia1
--    (("Ana","Lopez","Ruiz"),(10,"Marzo",1952),("Medica",100))
-- ---------------------------------------------------------------------

madreFamilia :: Familia -> Persona
madreFamilia (_,m,_) = m

-- ---------------------------------------------------------------------
-- Ejercicio 13.3 Definir la función
--    hijosFamilia :: Familia -> [Persona]
-- tal que (hijosFamilia f) es la lista de los hijos de la familia
-- f. Por ejemplo,
```

```
--    Main> hijosFamilia familia1
--    [(("Juan","Garcia","Lopez"),(5,"Enero",1970),("Estudiante",0)),
--    (("Maria","Garcia","Lopez"),(12,"Abril",1972),("Estudiante",0))]
-- ---------------------------------------------------------------------

hijosFamilia :: Familia -> [Persona]
hijosFamilia (_,_,hs) = hs

-- ---------------------------------------------------------------------
-- Ejercicio 13.4 Consultar si existe alguna familia sin hijos.
-- ---------------------------------------------------------------------

-- La consulta es
--    Main> not (null [f | f <- familias, null (hijosFamilia f)])
--    False
-- Luego, no existe familias sin hijos

-- ---------------------------------------------------------------------
-- Ejercicio 13.5 Consultar si existe alguna familia con tres hijos.
-- ---------------------------------------------------------------------

-- La consulta es
--    Main> not (null [f | f <- familias, length(hijosFamilia f)==3])
--    True
-- Luego, existen familias con tres hijos

-- ---------------------------------------------------------------------
-- Ejercicio 13.6 Consultar si existe alguna familia con cuatro hijos.
-- ---------------------------------------------------------------------

-- La consulta es
--    Main> not (null [f | f <- familias, length(hijosFamilia f)==4])
--    False
-- Luego, no existen familias con cuatro hijos

-- ---------------------------------------------------------------------
-- Ejercicio 13.7 Definir la función
--    nombre :: Persona -> NombreApellidos
-- tal que (nombre p) es el nombre de la persona p.
-- ---------------------------------------------------------------------
```

```
nombre :: Persona -> NombreApellidos
nombre (n,_,_) = n

-- ---------------------------------------------------------------------
-- Ejercicio 13.8 Calcular los nombres de los padres de las familias
-- con tres hijos.
-- ---------------------------------------------------------------------

-- La consulta es
--    Main> [nombre(padreFamilia f)
--           | f <- familias, length(hijosFamilia f)==3]
--    [("Jose","Perez","Ruiz")]

-- ---------------------------------------------------------------------
-- Ejercicio 13.9 Definir las constantes padres, madres e hijos que
-- calcule las listas de los padres, las madres y los hijos
-- respectivamente.
-- ---------------------------------------------------------------------

padres :: [Persona]
padres = [x | (x,y,z) <- familias]

madres :: [Persona]
madres = [y | (x,y,z) <- familias]

hijos :: [Persona]
hijos = concat [z | (x,y,z) <- familias]

-- Los valores son
--    Main> padres
--    [(("Tomas","Garcia","Perez"),(7,"Mayo",1950),("Profesor",75)),
--     (("Jose","Perez","Ruiz"),(6,"Marzo",1953),("Pintor",150))]
--    Main> madres
--    [(("Ana","Lopez","Ruiz"),(10,"Marzo",1952),("Medica",100)),
--     (("Luisa","Perez","Galvez"),(12,"Mayo",1954),("M\233dica",100))]
--    Main> hijos
--    [(("Juan","Garcia","Lopez"),(5,"Enero",1970),("Estudiante",0)),
--     (("Maria","Garcia","Lopez"),(12,"Abril",1972),("Estudiante",0)),
--     (("Juan Luis","Perez","Perez"),(5,"Febrero",1980),("Estudiante",0)),
```

```
--     (("Maria Jose","Perez","Perez"),(12,"Junio",1982),("Estudiante",0)),
--     (("Jose Maria","Perez","Perez"),(12,"Julio",1984),("Estudiante",0))]

-- ---------------------------------------------------------------------
-- Ejercicio 13.10 Definir la constante personas cuyo valor sea la lista
-- de todas las personas en la base de datos.
-- ---------------------------------------------------------------------

personas :: [Persona]
personas = padres ++ madres ++ hijos

-- ---------------------------------------------------------------------
-- Ejercicio 13.11 Hacer una pregunta tal que que la respuesta sean la
-- lista de elementos de la forma (nombre, apellido1, apellido2) de
-- todas las personas de la base de datos.
-- ---------------------------------------------------------------------

--    Main> [n | (n,f,t) <- personas]
--    [("Tomas","Garcia","Perez"),
--     ("Jose","Perez","Ruiz"),
--     ("Ana","Lopez","Ruiz"),
--     ("Luisa","Perez","Galvez"),
--     ("Juan","Garcia","Lopez"),
--     ("Maria","Garcia","Lopez"),
--     ("Juan Luis","Perez","Perez"),
--     ("Maria Jose","Perez","Perez"),
--     ("Jose Maria","Perez","Perez")]

-- ---------------------------------------------------------------------
-- Ejercicio 13.12 Definir la función
--    profesión :: Persona -> Profesión
-- tal que (profesión x) es la profesión de la persona x.
-- ---------------------------------------------------------------------

profesión :: Persona -> Profesión
profesión (_,_,(p,_)) = p

-- ---------------------------------------------------------------------
-- Ejercicio 13.13 Definir la función
--    añoNacimiento :: Persona -> Año
```

```
-- tal que (añoNacimiento x) es el año de nacimiento de la persona x.
-- ---------------------------------------------------------------------

añoNacimiento :: Persona -> Año
añoNacimiento (_,(_,_,a),_) = a

-- ---------------------------------------------------------------------
-- Ejercicio 13.14 Determinar todos los estudiantes nacidos antes de
-- 1983.
-- ---------------------------------------------------------------------

--    Main> [p | p <- personas
--             , profesión(p)=="Estudiante"
--             , añoNacimiento(p)<1983]
--    [(("Juan","Garcia","Lopez"),(5,"Enero",1970),("Estudiante",0)),
--     (("Maria","Garcia","Lopez"),(12,"Abril",1972),("Estudiante",0)),
--     (("Juan Luis","Perez","Perez"),(5,"Febrero",1980),("Estudiante",0)),
--     (("Maria Jose","Perez","Perez"),(12,"Junio",1982),("Estudiante",0))]

-- ---------------------------------------------------------------------
-- Ejercicio 13.15 Definir la función
--    fecha_de_nacimiento :: Persona -> Fecha
-- tal que (fecha_de_nacimiento x) es la fecha de nacimiento de la
-- persona x.
-- ---------------------------------------------------------------------

fecha_de_nacimiento :: Persona -> Fecha
fecha_de_nacimiento (_,f,_) = f

-- ---------------------------------------------------------------------
-- Ejercicio 13.16 Buscar todos los hijos nacidos en 1982.
-- ---------------------------------------------------------------------

--    Main> [x | x <- hijos, añoNacimiento(x)==1982]
--    [(("Maria Jose","Perez","Perez"),(12,"Junio",1982),("Estudiante",0))]

-- ---------------------------------------------------------------------
-- Ejercicio 13.17 Definir la función
--    sueldo :: Persona -> Int
-- tal que (sueldo x) es el sueldo de la persona x.
```

```
-- ---------------------------------------------------------------------

sueldo :: Persona -> Int
sueldo (_,_,(_,s)) = s

-- ---------------------------------------------------------------------
-- Ejercicio 13.18 Buscar todas las personas nacidas antes de 1954 cuyo
-- sueldo sea superior a 72 euros.
-- ---------------------------------------------------------------------

--    Main> [x | x <- personas, añoNacimiento(x)<1954, sueldo(x)>72]
--    [(("Tomas","Garcia","Perez"),(7,"Mayo",1950),("Profesor",75)),
--     (("Jose","Perez","Ruiz"),(6,"Marzo",1953),("Pintor",150)),
--     (("Ana","Lopez","Ruiz"),(10,"Marzo",1952),("Medica",100))]

-- ---------------------------------------------------------------------
-- Ejercicio 13.19 Definir la función
--    total :: [Persona] -> Int
-- tal que (total xs) es la suma de los sueldos de las personas de la
-- lista xs.
-- ---------------------------------------------------------------------

total :: [Persona] -> Int
total xs = sum [sueldo(x) | x <- xs]

-- ---------------------------------------------------------------------
-- Ejercicio 13.20 Calcular los ingresos totales de cada familia.
-- ---------------------------------------------------------------------

totalFamilia :: Familia -> Int
totalFamilia (p,m,hs) = (sueldo p)+(sueldo m)+(total hs)

--    Main> [(nombre(padreFamilia(f)),totalFamilia(f)) | f <- familias]
--    [(("Tomas","Garcia","Perez"),175),(("Jose","Perez","Ruiz"),250)]

-- ---------------------------------------------------------------------
-- Ejercicio 14 (Simulación del autómata)
-- ---------------------------------------------------------------------
-- En este ejercicio consideraremos la versión Haskell del programa
-- automata.pl del Tema
```

```
-- 8. El autómata se define mediante las siguientes relaciones
--  * final(E) que se verifica si E es el estado final.
--  * trans(E1,X,E2) que se verifica si se puede pasar del estado E1 al
--    estado E2 usando la letra X.
--  * nulo(E1,E2) que se verifica si se puede pasar del estado E1 al
--    estado E2 mediante un movimiento nulo.
-- El autómata del ejemplo está definido por
data Estado = E1 | E2 | E3 | E4
             deriving (Show, Eq)

estados :: [Estado]
estados = [E1, E2, E3, E4]

type Acción = Char
acciones :: [Acción]
acciones =['a', 'b']

finales :: [Estado]
finales = [E3]

trans :: Estado -> Acción -> [Estado]
trans E1 'a' = [E1,E2]
trans E1 'b' = [E1]
trans E2 'b' = [E3]
trans E3 'b' = [E4]
trans _  _   = []

nulo :: Estado -> [Estado]
nulo E2  = [E4]
nulo E3  = [E1]
nulo _   = []
-- ----------------------------------------------------------------------

-- ----------------------------------------------------------------------
-- Ejercicio 14.1 Definir la función
--     final :: Estado -> Bool
-- tal que (final e) se verifica si e es un estado final.
-- ----------------------------------------------------------------------

final :: Estado -> Bool
```

```
final x =
    elem x finales

-- ---------------------------------------------------------------------
-- Ejercicio 14.2 Definir la función
--    acepta :: Estado -> String -> Bool
-- tal que (acepta e c) se verifica si el autómata, a partir del estado
-- e acepta la cadena c. Por ejemplo,
--    acepta E1 "aaab"  ==>  True
--    acepta E2 "aaab"  ==>  False
--    acepta E3 "ab"    ==>  True
-- ---------------------------------------------------------------------

acepta :: Estado -> String -> Bool
acepta e []     = final e
acepta e (c:cs) = or ([acepta e1 cs | e1 <- trans e c] ++
                      [acepta e1 (c:cs) | e1 <- nulo e])
acepta _ _      = False

-- ---------------------------------------------------------------------
-- Ejercicio 14.3 Determinar los estados a partir de los cuales el
-- autómata acepta la cadena "ab".
-- ---------------------------------------------------------------------

-- La consulta es
--    Main> [e | e <- estados, acepta e "ab"]
--    [E1,E3]

-- ---------------------------------------------------------------------
-- Ejercicio 14.4 Determinar las palabras de longitud 3 aceptadas por el
-- autómata a partir del estado E1.
-- ---------------------------------------------------------------------

-- La consulta es
--    Main> [[x,y,z] | x <- acciones, y <- acciones, z <- acciones,
--                     acepta E1 [x,y,z]]
--    ["aab","bab"]

-- ---------------------------------------------------------------------
-- Ejercicio 14.5 Definir la función
```

```
--    aceptadas_con_longitud :: Estado -> Int -> [[Acción]]
-- tal que (aceptadas_con_longitud e n) es la lista de palabras de
-- longitud n aceptadas por el autómata a partir del estado E. Por
-- ejemplo,
--    aceptadas_con_longitud E1 3  ==>  ["aab","bab"]
-- ----------------------------------------------------------------------

aceptadas_con_longitud :: Estado -> Int -> [[Acción]]
aceptadas_con_longitud e n =
    [c | c <- palabras n, acepta e c]

-- (palabras n) es la lista de palabras de longitud n. Por ejemplo,
--    palabras 2  ==>  ["aa","ab","ba","bb"]
palabras :: Int -> [[Acción]]
palabras 0     = [[]]
palabras (n+1) = [(x:xs) | x <- acciones, xs <- palabras n]

-- ----------------------------------------------------------------------
-- Ejercicio 14.6 Definir la función
--    aceptadas_con_cota :: Estado -> Int -> [[Acción]]
-- tal que (aceptadas_con_cota e n) es la lista de palabras de
-- longitud menor o igual que n aceptadas por el autómata a partir del
-- estado E. Por ejemplo,
--    aceptadas_con_cota E1 3  ==>  ["ab","aab","bab"]
-- ----------------------------------------------------------------------

aceptadas_con_cota :: Estado -> Int -> [[Acción]]
aceptadas_con_cota e n =
    [c | m <- [0..n], c <- palabras m, acepta e c]

-- ----------------------------------------------------------------------
-- Ejercicio 15 (Movimientos del caballo)
-- ----------------------------------------------------------------------
-- Supongamos que los cuadros del tablero de ajedrez los representamos
-- por pares de números [X,Y] con X e Y entre 1 y 8.
-- ----------------------------------------------------------------------
-- Ejercicio 15.1 Definir la función
--    saltos :: Cuadrado -> [Cuadrado]
-- tal que (saltos c) es la lista de cuadrados a donde el caballo puede
-- pasar en un movimiento a partir del cuadrado c. Por ejemplo,
```

```
--    saltos (1,1)  ==>  [(3,2),(2,3)]
-- ---------------------------------------------------------------------

type Cuadrado = (Int,Int)

saltos :: Cuadrado -> [Cuadrado]
saltos (x,y) =
    [(x+dx,y+dy) | (dx,dy) <- movimientos
                 , correcto (x+dx)
                 , correcto (y+dy)]

movimientos :: [(Int,Int)]
movimientos = [(2,1), (2,-1), (-2,1), (-2,-1),
               (1,2), (1,-2), (-1,2), (-1,-2)]

-- (correcto x se verifica si x está entre 1 y 8.
correcto x = 1 <= x && x <= 8

-- ---------------------------------------------------------------------
-- Ejercicio 15.2 Definir la
--    camino :: [Cuadrado] -> Bool
-- tal que (camino cs) se verifica si cs es una lista de cuadrados
-- representando el camino recorrido por un caballo sobre un tablero
-- vacío. Por ejemplo,
--    camino [(1,1),(3,2),(1,1)]  ==>  True
--    camino [(1,1),(3,2),(1,2)]  ==>  False
-- ---------------------------------------------------------------------

camino :: [Cuadrado] -> Bool
camino [_] = True
camino (c1:c2:cs) = c2 `elem` saltos c1 && camino (c2:cs)

-- ---------------------------------------------------------------------
-- Ejercicio 15.3 Usando la relación camino, escribir una pregunta para
-- determinar los caminos de longitud 4 por los que puede desplazarse un
-- caballo desde cuadro (2,1) hasta el otro extremo del tablero (Y=8) de
-- forma que en el segundo movimiento pase por el cuadro (5,4).
-- ---------------------------------------------------------------------

cuadrados :: [Cuadrado]
```

```
cuadrados = [(x,y) | x <- [1..8], y <- [1..8]]

-- Main> [[(2,1),c1,(5,4),c2,(x,8)] | c1 <- cuadrados
--                                  , c2 <- cuadrados
--                                  , x <- [1..8]
--                                  , camino [(2,1),c1,(5,4),c2,(x,8)]]
-- [[(2,1),(3,3),(5,4),(4,6),(3,8)],
--  [(2,1),(3,3),(5,4),(4,6),(5,8)],
--  [(2,1),(3,3),(5,4),(6,6),(5,8)],
--  [(2,1),(3,3),(5,4),(6,6),(7,8)],
--  [(2,1),(4,2),(5,4),(4,6),(3,8)],
--  [(2,1),(4,2),(5,4),(4,6),(5,8)],
--  [(2,1),(4,2),(5,4),(6,6),(5,8)],
--  [(2,1),(4,2),(5,4),(6,6),(7,8)]]
-- Main> [[(2,1),c1,(5,4),c2,(x,8)] | c1 <- saltos (2,1)
--                                  , elem (5,4) (saltos c1)
--                                  , c2 <- saltos (5,4)
--                                  , x <- [1..8]
--                                  , elem (x,8) (saltos c2)]
-- [[(2,1),(4,2),(5,4),(6,6),(5,8)],
--  [(2,1),(4,2),(5,4),(6,6),(7,8)],
--  [(2,1),(4,2),(5,4),(4,6),(3,8)],
--  [(2,1),(4,2),(5,4),(4,6),(5,8)],
--  [(2,1),(3,3),(5,4),(6,6),(5,8)],
--  [(2,1),(3,3),(5,4),(6,6),(7,8)],
--  [(2,1),(3,3),(5,4),(4,6),(3,8)],
--  [(2,1),(3,3),(5,4),(4,6),(5,8)]]

-- ------------------------------------------------------------------
-- Ejercicio 15.4 Calcular el menor número de movimientos necesarios
-- para desplazar el caballo desde el cuadrado (1,1) hasta el
-- (2,2). ¿Cuántos caminos de dicha longitud hay de (1,1) a (2,2)?
-- ------------------------------------------------------------------

-- Main> [[(1,1),c1,(2,2)]
--        | c1 <- cuadrados
--        , camino [(1,1),c1,(2,2)]]
-- []
-- Main> [[(1,1),c1,c2,(2,2)]
--        | c1 <- cuadrados
```

```
--         , c2 <- cuadrados
--         , camino [(1,1),c1,c2,(2,2)]]
-- []
-- Main> [[(1,1),c1,c2,c3,(2,2)]
--         | c1 <- cuadrados
--         , c2 <- cuadrados
--         , c3 <- cuadrados
--         , camino [(1,1),c1,c2,c3,(2,2)]]
-- [[(1,1),(2,3),(1,5),(3,4),(2,2)],
--  [(1,1),(2,3),(3,1),(4,3),(2,2)],
--  [(1,1),(2,3),(3,5),(1,4),(2,2)],
--  [(1,1),(2,3),(3,5),(4,3),(2,2)],
--  [(1,1),(2,3),(4,2),(3,4),(2,2)],
--  [(1,1),(3,2),(1,3),(3,4),(2,2)],
--  [(1,1),(3,2),(2,4),(4,3),(2,2)],
--  [(1,1),(3,2),(5,1),(4,3),(2,2)],
--  [(1,1),(3,2),(5,3),(3,4),(2,2)],
--  [(1,1),(3,2),(5,3),(4,1),(2,2)]]

-- ---------------------------------------------------------------------
-- Ejercicio 16 (Problema del mono)
-- ---------------------------------------------------------------------

data Sitio = Puerta | Centro | Ventana deriving (Eq, Show)
type PM = Sitio
data AM = Suelo | Silla deriving (Eq, Show)
type PS = Sitio
data MM = Con | Sin deriving (Eq, Show)

data Estado' = E PM AM PS MM deriving (Eq, Show)

data Acción' = Coger
             | Subir
             | Empujar Sitio Sitio
             | Pasear Sitio Sitio
               deriving (Eq, Show)

movimiento :: Estado' -> Acción' -> [Estado']
movimiento (E Centro Silla Centro Sin) Coger
    = [E Centro Silla Centro Con]
```

```
movimiento (E x Suelo y u) Subir
    | x==y = [E x Silla x u]
movimiento (E x1 Suelo y1 u) (Empujar z1 z2)
    | x1== y1 && x1==z1 = [E z2 Suelo z2 u]
movimiento (E x1 Suelo z1 u) (Pasear x2 z2)
    | x1==x2 && z1==z2 = [E z1 Suelo z1 u]
movimiento _ _ = []

sitios :: [Sitio]
sitios = [Puerta, Centro, Ventana]

acciones' :: [Acción']
acciones' = [Coger, Subir] ++
            [Empujar x y | x <- sitios, y <- sitios] ++
            [Pasear x y | x <- sitios, y <- sitios]

--    Main> sucesores (E Puerta Suelo Ventana Sin)
--    [E Ventana Suelo Ventana Sin]
sucesores :: Estado' -> [Estado']
sucesores e = concat [movimiento e a | a <- acciones']

esFinal :: Estado' -> Bool
esFinal (E _ _ _ x) = x==Con

--    Main> solución (E Puerta Suelo Ventana Sin)
--    [E Puerta  Suelo Ventana Sin,
--     E Ventana Suelo Ventana Sin,
--     E Puerta  Suelo Puerta  Sin,
--     E Centro  Suelo Centro  Sin,
--     E Centro  Silla Centro  Sin,
--     E Centro  Silla Centro  Con]
solución :: Estado' -> [Estado']
solución e = reverse(head(soluciones e []))

soluciones :: Estado' -> [Estado'] -> [[Estado']]
soluciones e vis
    | esFinal e = [e:vis]
    | otherwise = concat [soluciones e' (e:vis)
                          | e' <- sucesores e
                          , notElem e' (e:vis)]
```

Capítulo 9

Práctica 4a (19 de diciembre de 2007)

9.1. Práctica 4a en Prolog

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 1 Definir la relación diferencia(C1,C2,C3) que se verifique
% si C3 es la diferencia de los conjuntos C1 y C2. Por ejemplo,
%    ?- diferencia([a,b],[b,c],X)
%    X = [a] ;
%    No
% Escribir una versión con not y otra con corte.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% (1) Versión con not:
diferencia_1([],_,[]).
diferencia_1([X|C1],C2,C3) :-
   member(X,C2),
   diferencia_1(C1,C2,C3).
diferencia_1([X|C1],C2,[X|C3]) :-
   not(member(X,C2)),
   diferencia_1(C1,C2,C3).

% (2) Versión con corte:
diferencia_2([],_,[]).
diferencia_2([X|C1],C2,C3) :-
   member(X,C2), !,
   diferencia_2(C1,C2,C3).
diferencia_2([X|C1],C2,[X|C3]) :-
   % not(member(X,C2)),
   diferencia_2(C1,C2,C3).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 2 Definir la relación agregar(X,L,L1) que se verifique si
% L1 es la lista obtenida añadiéndole X a L, si X no pertenece a L y es
% L en caso contrario. Por ejemplo,
%    ?- agregar(a,[b,c],L).
%    L = [a,b,c]
%    ?- agregar(b,[b,c],L).
%    L = [b,c]
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Primera definición:
agregar_1(X,L,[X|L]) :-
   not(member(X,L)).
agregar_1(X,L,L) :-
   member(X,L).

% Segunda definición:
agregar_2(X,L,L) :-
   memberchk(X,L), !.
agregar_2(X,L,[X|L]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 3 Definir la relación separa(L1,L2,L3) que separa la lista
% de números L1 en dos listas: L2 formada por los números positivos y L3
% formada por los números negativos o cero. Por ejemplo,
%    ?- separa([2,0,-3,5,0,2],L2,L3).
%    L2 = [2, 5, 2]
%    L3 = [0, -3, 0]
%    Yes
% Proponer dos soluciones, una sin corte y otra con corte.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Primera solución:
separa_1([],[],[]).
separa_1([N|RL1],[N|RL2],L3) :-
   N > 0,
   separa_1(RL1,RL2,L3).
separa_1([N|RL1],L2,[N|RL3]) :-
   N =< 0,
```

```
   separa_1(RL1,L2,RL3).

% Segunda solución:
separa_2([],[],[]).
separa_2([N|RL1],[N|RL2],L3) :-
   N > 0, !,
   separa_2(RL1,RL2,L3).
separa_2([N|RL1],L2,[N|RL3]) :-
   % N =< 0,
   separa_2(RL1,L2,RL3).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 4 Definir la relación suma_pares(L,N) que se verifica si N
% es la suma de todos los números pares de la lista L. Por ejemplo,
%    ?- suma_pares([2,3,4],N).
%    N = 6
%    ?- suma_pares([1,3,5,6,9,11,24],N).
%    N = 30
% Escribir dos definiciones, una versión con negación y otra con corte.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Versión 1 (con negación):
suma_pares_1([],0).
suma_pares_1([N|L],X) :-
   par(N),
   suma_pares_1(L,X1),
   X is X1 + N.
suma_pares_1([_N|L],X) :-
   not(par(_N)),
   suma_pares_1(L,X).

par(N):-
   N mod 2 =:= 0.

% Versión 2 (con corte):
suma_pares_2([],0).
suma_pares_2([N|L],X) :-
   par(N), !,
   suma_pares_2(L,X1),
   X is X1 + N.
```

```
suma_pares_2([_N|L],X) :-
   % not(par(_N)),
   suma_pares_2(L,X).

% Versión 3 (con corte y acumulador):
suma_pares_3(L,X):-
   suma_pares_3_aux(L,0,X).

suma_pares_3_aux([],Ac,Ac).
suma_pares_3_aux([N|L],Ac,X) :-
   par(N), !,
   Ac1 is Ac + N,
   suma_pares_3_aux(L,Ac1,X).
suma_pares_3_aux([_N|L],Ac,X) :-
   % not(par(_N)),
   suma_pares_3_aux(L,Ac,X).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 5 Definir la relación exponente_de_dos(N,Exp) que se
% verifique si E es el exponente de 2 en la descomposición de N como
% producto de factores primos. Por ejemplo,
%    ?- exponente_de_dos(40,E).
%    E=3
%    ?- exponente_de_dos(49,E).
%    E=0
% Proponer una versión con negación y otra con corte.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% 1ª Versión (con negación):
exponente_de_dos_1(N,E):-
   N mod 2 =:= 0,
   N1 is N / 2,
   exponente_de_dos_1(N1,E1),
   E is E1 + 1.
exponente_de_dos_1(N,0) :-
   N mod 2 =\= 0.

% 2ª Versión (con corte):
exponente_de_dos_2(N,E):-
   N mod 2 =:= 0, !,
```

```
   N1 is N / 2,
   exponente_de_dos_2(N1,E1),
   E is E1 + 1.
exponente_de_dos_2(_,0).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 6 Definir la relación lista_a_conjunto(+L,-C) que se
% verifique si C es el conjunto correspondiente a la lista L (es decir,
% C contiene los mismos elementos que L en el mismo orden, pero si L
% tiene elementos repetidos sólo se incluye en C la última aparición de
% cada elemento). Por ejemplo,
%    ?- lista_a_conjunto([b,a,b,d],C).
%    C = [a, b, d]
% Nota: La relación lista_a_conjunto se corresponde con la relación
% predefinida list_to_set.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% 1ª definición (con negación):
lista_a_conjunto([],[]).
lista_a_conjunto([X|L],C) :-
   member(X,L),
   lista_a_conjunto(L,C).
lista_a_conjunto([X|L],[X|C]) :-
   \+ member(X,L),
   lista_a_conjunto(L,C).

% 2ª definición (con corte):
lista_a_conjunto_1([],[]).
lista_a_conjunto_1([X|L],C) :-
   member(X,L), !,
   lista_a_conjunto_1(L,C).
lista_a_conjunto_1([X|L],[X|C]) :-
   % \+ member(X,L),
   lista_a_conjunto_1(L,C).

% 3ª definición (con corte y memberchk):
lista_a_conjunto_3([],[]).
lista_a_conjunto_3([X|L],L2):-
   memberchk(X,L), !,
   lista_a_conjunto_3(L,L2).
```

```
lista_a_conjunto_3([X|L],[X|L2]):-
   % not(member(X,L)),
   lista_a_conjunto_3(L,L2).

% Comparaciones:
%   ?- numlist(1,1000,_L1), time(lista_a_conjunto_1(_L1,_l2)).
%   % 1,003,001 inferences, 0,40 CPU in 0,41 seconds (97% CPU, 2507503 Lips)
%   ?- numlist(1,1000,_L1), time(lista_a_conjunto_2(_L1,_l2)).
%   % 501,501 inferences, 0,26 CPU in 0,28 seconds (93% CPU, 1928850 Lips)
%   ?- numlist(1,1000,_L1), time(lista_a_conjunto_3(_L1,_l2)).
%   % 3,001 inferences, 0,07 CPU in 0,08 seconds (90% CPU, 42871 Lips)
%   ?- numlist(1,1000,_L1), time(list_to_set(_L1,_l2)).
%   % 3,004 inferences, 0,07 CPU in 0,08 seconds (93% CPU, 42914 Lips)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 7 Definir la relación crecimientos(+L1,-L2) que se verifique
% si L2 es la lista correspondientes a los crecimientos de la lista
% numérica L1; es decir, entre cada par de elementos consecutivos X e Y
% de L1 coloca el signo + si X < Y e y signo - en caso contrario. Por
% ejemplo,
%    ?- crecimientos([1,3,2,2,5,3],L).
%    L = [1, +, 3, -, 2, -, 2, +, 5, -]
% Dar una definición sin corte y otra con corte.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% 1ª definición (sin cortes):
crecimientos_1([_],[]).
crecimientos_1([X,Y|L1],[X,+|L2]) :-
   X < Y,
   crecimientos_1([Y|L1],L2).
crecimientos_1([X,Y|L1],[X,-|L2]) :-
   X >= Y,
   crecimientos_1([Y|L1],L2).

% 2ª definición (con cortes):
crecimientos_2([_],[]).
crecimientos_2([X,Y|L1],[X,+|L2]) :-
   X < Y, !,
   crecimientos_2([Y|L1],L2).
crecimientos_2([X,Y|L1],[X,-|L2]) :-
```

```
   % X >= Y,
   crecimientos_2([Y|L1],L2).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 8 Definir las siguientes relaciones:
%  * menor_divisor_propio(+N,?X) que se verifique si X es el menor
%    divisor de N mayor o igual que 2. Por ejemplo,
%       ?- menor_divisor_propio(30,X).
%       X = 2
%       ?- menor_divisor_propio(3,X).
%       X = 3
%  * factorización(+N,-L) que se verifique si L es la lista
%    correspondiente a la descomposición del número N en factores primos
%    (se considera que lo elementos de L están ordenados de manera
%    creciente). Por ejemplo,
%       ?- factorización(12,L).
%       L = [2, 2, 3] ;
%       No
%       ?- factorización(1,L).
%       L = [] ;
%       No
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% La definición de menor_divisor_propio es
menor_divisor_propio(N,X) :-
   N1 is floor(sqrt(N)),
   between(2,N1,X),
   N mod X =:= 0, !.
menor_divisor_propio(N,N).

% La definición de factorización es
factorización(1,[]).
factorización(N,[X|L]) :-
   N > 1,
   menor_divisor_propio(N,X),
   N1 is N/X,
   factorización(N1,L).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 9 Definir la relación calcula(+N,+M,?X) que se verifique si
```

```
% X es el menor múltiplo de N tal que la suma de sus dígitos es mayor
% que M. Por ejemplo,
%    ?- calcula(3,10,X).
%    X = 39
%    Yes
%    ?- calcula(7,20,X).
%    X = 399
%    Yes
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% La definición de calcula es
calcula(N,M,X) :-
   múltiplo(N,X),
   suma_dígitos(X,N1),
   N1 > M, !.

% La relación múltiplo(+N,-X) se verifica si X es un múltiplo de N. Por
% ejemplo,
%   ?- múltiplo(5,X).
%   X = 5 ;
%   X = 10 ;
%   X = 15
%   Yes
múltiplo(N,N).
múltiplo(N,M) :-
   múltiplo(N,N1),
   M is N+N1.

% La relación suma_dígitos(+N,-S) se verifica si S es la suma de los
% dígitos del número N. Por ejemplo,
%   ?- suma_dígitos(237,S).
%   S = 12
suma_dígitos(N,N) :-
   N < 10, !.
suma_dígitos(N,S) :-
   % N >= 10,
   N1 is N // 10,
   R is N - 10*N1,
   suma_dígitos(N1,S1),
   S is S1 + R.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 10 Un número es libre de cuadrados si no es divisible por el
% cuadrado de ningún número mayor que 1. Definir la relación
% libre_de_cuadrados(+N) que se verifique si el número N es libre de
% cuadrados. Por ejemplo,
%    ?- libre_de_cuadrados(30).
%    Yes
%    ?- libre_de_cuadrados(12).
%    No
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

libre_de_cuadrados(N) :-
   M is floor(sqrt(N)),
   not((between(2,M,X), N mod (X*X) =:= 0)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 11 Definir la relación suma_libres_de_cuadrados(+L,-S) que
% se verifique si S es la suma de los números libres de cuadrados la
% lista numérica L. Por ejemplo,
%    ?- suma_libres_de_cuadrados([6,12,18,30],S).
%    S = 36
% Nota: Dar dos definiciones, una con negación y otra con corte.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% 1ª definición (con negación):
suma_libres_de_cuadrados_1([],0).
suma_libres_de_cuadrados_1([X|L],S) :-
   libre_de_cuadrados(X),
   suma_libres_de_cuadrados_1(L,S1),
   S is X+S1.
suma_libres_de_cuadrados_1([X|L],S) :-
   not(libre_de_cuadrados(X)),
   suma_libres_de_cuadrados_1(L,S).

% 2ª definición (con corte):
suma_libres_de_cuadrados_2([],0).
suma_libres_de_cuadrados_2([X|L],S) :-
   libre_de_cuadrados(X), !,
   suma_libres_de_cuadrados_2(L,S1),
```

```
   S is X+S1.
suma_libres_de_cuadrados_2([_X|L],S) :-
   % not(libre_de_cuadrados(_X)),
   suma_libres_de_cuadrados_2(L,S).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 12 Definir la relación max_lista(+L,-N) que se verifique si
% N es el mayor número de la lista L. Por ejemplo,
%    ?- max_lista([2,a23,5,7+9],N).
%    N = 5
%    ?- max_lista([-2,a23,-5,7+9],N).
%    N = -2
%    ?- max_lista([a23,7+9],N).
%    No
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

max_lista(L,M) :-
   member(M,L),
   number(M),
   not((member(N,L),
        number(N),
        N>M)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 13 Definir la relación longitud_scm(+L1,+L2,-N) que se
% verifique si N es la longitud de las subsucesiones comunes maximales
% de las listas L1 y L2. Por ejemplo,
%    ?- longitud_scm([2,1,4,5,2,3,5,2,4,3],[1,7,5,3,2],N).
%    N = 4 ;
%    No
% ya que [1,5,3,2] es una subsucesión de las dos listas y no poseen
% ninguna otra subsucesión común de mayor longitud. Obsérvese que los
% elementos de la subsucesión no son necesariamente elementos adyacentes
% en las listas.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% La definición de longitud_scm es
longitud_scm([],_,0).
longitud_scm(_,[],0).
longitud_scm([X|L1],[X|L2],N) :-
```

```
   !, longitud_scm(L1,L2,M),
   N is M+1.
longitud_scm([X|L1],[Y|L2],N) :-
   % X \= Y,
   longitud_scm(L1,[Y|L2],N1),
   longitud_scm([X|L1],L2,N2),
   N is max(N1,N2).
```

9.2. Práctica 4a en Haskell

```
-- ---------------------------------------------------------------------
-- Librerías auxiliares
-- ---------------------------------------------------------------------

import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1 Definir la función
--    diferencia :: Eq a => [a] -> [a] -> [a]
-- tal que (diferencia xs ys) es la diferencia de los conjuntos xs e
-- ys. Por ejemplo,
--    diferencia [1,2] [2,3]  ==>  [1]
-- Proponer dos soluciones, una recursiva y otra por comprensión.
-- ---------------------------------------------------------------------

-- 1ª definición (recursiva):
diferencia_1 :: Eq a => [a] -> [a] -> [a]
diferencia_1 [] _ = []
diferencia_1 (x:xs) ys
    | elem x ys = diferencia_1 xs ys
    | otherwise = x:(diferencia_1 xs ys)

-- 2ª definición (por comprensión):
diferencia_2 :: Eq a => [a] -> [a] -> [a]
diferencia_2 xs ys = [x | x <- xs, notElem x ys]

-- ---------------------------------------------------------------------
-- Ejercicio 2 Definir la función
--    agregar :: Eq a => a -> [a] -> [a]
-- tal que (agregar x ys) es la lista obtenida añadiéndole x a ys, si x
```

```
-- no pertenece a ys y es ys en caso contrario. Por ejemplo,
--    agregar 1 [2,3]  ==>  [1,2,3]
--    agregar 2 [2,3]  ==>  [2,3]
-- ---------------------------------------------------------------------

agregar :: Eq a => a -> [a] -> [a]
agregar x ys
    | elem x ys = ys
    | otherwise = x:ys

-- ---------------------------------------------------------------------
-- Ejercicio 3 Definir la función
--    separa :: [Int] -> ([Int],[Int])
-- tal que (separa xs) separa la lista de números xs en dos listas: una
-- formada por los números positivos y la otra formada por los números
-- negativos o cero. Por ejemplo,
--    separa [2,0,-3,5,0,2]  ==>  ([2,5,2],[0,-3,0])
-- Proponer dos soluciones, una recursiva y otra por comprensión.
-- ---------------------------------------------------------------------

-- 1ª definición (recursiva):
separa_1 :: [Int] -> ([Int],[Int])
separa_1 [] = ([],[])
separa_1 (x:xs)
    | x>0       = (x:ys,zs)
    | otherwise = (ys,x:zs)
    where (ys,zs) = separa_1 xs

-- 2ª definición (por comprensión):
separa_2 :: [Int] -> ([Int],[Int])
separa_2 xs = ([x | x <- xs, x>0], [x | x <- xs, x<=0])

-- ---------------------------------------------------------------------
-- Ejercicio 4 Definir la función
--    suma_pares :: [Int] -> Int
-- tal que (suma_pares xs) es la suma de todos los números pares de la
-- lista xs. Por ejemplo,
--    suma_pares_1 [2,3,4]            ==>  6
--    suma_pares_1 [1,3,5,6,9,11,24]  ==>  30
-- Escribir dos definiciones, una recursiva y otra por comprensión.
```

```
-- ---------------------------------------------------------------------

-- 1ª definición (recursiva):
suma_pares_1 :: [Int] -> Int
suma_pares_1 [] = 0
suma_pares_1 (x:xs)
    | even x    = x+(suma_pares_1 xs)
    | otherwise = suma_pares_1 xs

-- 2ª definición (por comprensión):
suma_pares_2 :: [Int] -> Int
suma_pares_2 xs = sum [x | x <- xs, even x]

-- ---------------------------------------------------------------------
-- Ejercicio 5 Definir la función
--    exponente_de_dos :: Int -> Int
-- tal que (exponente_de_dos n) es el exponente de 2 en la
-- descomposición de n como producto de factores primos. Por ejemplo,
--    exponente_de_dos 40  ==>  3
--    exponente_de_dos 49  ==>  0
-- ---------------------------------------------------------------------

exponente_de_dos :: Int -> Int
exponente_de_dos n
    | n `mod` 2 == 0 = 1+(exponente_de_dos (n `div` 2))
    | otherwise    = 0

-- ---------------------------------------------------------------------
-- Ejercicio 6 Definir la función
--    lista_a_conjunto :: Eq a => [a] -> [a]
-- tal que (lista_a_conjunto xs) es el conjunto correspondiente a la
-- lista xs (es decir, contiene los mismos elementos que xs en el mismo
-- orden, pero si xs tiene elementos repetidos sólo se incluye la última
-- aparición de cada elemento). Por ejemplo,
--    lista_a_conjunto [2,1,2,0]  ==> [1,2,0]
-- Nota: La función lista_a_conjunto se corresponde con la función nub
-- definida en la librería Data.List.
-- ---------------------------------------------------------------------

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

```
lista_a_conjunto [] = []
lista_a_conjunto (x:xs)
    | elem x xs = lista_a_conjunto xs
    | otherwise = x:(lista_a_conjunto xs)

-- ---------------------------------------------------------------------
-- Ejercicio 7 Definir la función
--    crecimientos :: [Int] -> [(Int,Char)]
-- tal que (crecimientos xs) es la lista correspondientes a los
-- crecimientos de la lista numérica xs; es decir, entre cada elemento
-- de xs se acompña del signo + ó - según que sea menor o no que su
-- siguiente elemento en la lista xs. Por
-- ejemplo,
--    Main> crecimientos [1,3,2,2,5,3]  ==>
--    [(1,'+'),(3,'-'),(2,'-'),(2,'+'),(5,'-')]
-- ---------------------------------------------------------------------

crecimientos :: [Int] -> [(Int,Char)]
crecimientos [x] = []
crecimientos (x:y:zs)
    | x<y       = (x,'+'):(crecimientos (y:zs))
    | otherwise = (x,'-'):(crecimientos (y:zs))

-- ---------------------------------------------------------------------
-- Ejercicio 8 Definir la función
--    menor_divisor_propio :: Integer -> Integer
-- tal que (menor_divisor_propio n) es el menor divisor de n mayor o
-- igual que 2. Por ejemplo,
--   menor_divisor_propio 30  ==>  2
--   menor_divisor_propio 3   ==>  3
-- y la función
--    factorización :: Integer -> [Integer]
-- tal que (factorización n) es la lista correspondiente a la
-- descomposición del número n en factores primos (se considera que lo
-- elementos del resultado están ordenados de manera creciente). Por
-- ejemplo,
--    factorización 12  ==>  [2,2,3]
--    factorización 1   ==>  []
-- ---------------------------------------------------------------------
```

```
menor_divisor_propio :: Integer -> Integer
menor_divisor_propio n
    | not(null xs) = head xs
    | otherwise    = n
    where xs = [x | x <- [2..floor(sqrt(fromInteger(n)))]
                  , n `mod` x == 0]

factorización :: Integer -> [Integer]
factorización 1       = []
factorización n | n>1 = x:(factorización (n `div` x))
                        where x = menor_divisor_propio n

-- ---------------------------------------------------------------------
-- Ejercicio 9 Definir la función
--    calcula :: Int -> Int -> Int
-- tal que (calcula n m) es el menor múltiplo de n tal que la suma de
-- sus dígitos es mayor que m. Por ejemplo,
--    calcula 3 10  ==>  39
--    calcula 7 20  ==>  399
-- ---------------------------------------------------------------------

calcula :: Int -> Int -> Int
calcula n m =
    head [x | x <- múltiplos n, suma_dígitos x > m]

-- (múltiplos n) es la lista de los múltiplos de n. Por ejemplo,
--    take 10 (múltiplos 3)  ==>  [3,6,9,12,15,18,21,24,27,30]
múltiplos :: Int -> [Int]
múltiplos n =
    [n*x | x <- [1..]]

-- (suma_dígitos n) es la suma de los dígitos del número n. Por ejemplo,
--   suma_dígitos 237  ==>  12
suma_dígitos :: Int -> Int
suma_dígitos n
    | n<10      = n
    | otherwise = (n-10*n1)+(suma_dígitos n1)
    where n1 = n `div` 10

-- ---------------------------------------------------------------------
```

```
-- Ejercicio 10 Un número es libre de cuadrados si no es divisible por el
-- cuadrado de ningún número mayor que 1. Definir la función
--    libre_de_cuadrados :: Int -> Bool
-- tal que (libre_de_cuadrados n) se verifique si el número n es libre de
-- cuadrados. Por ejemplo,
--    libre_de_cuadrados 30  ==>  True
--    libre_de_cuadrados 12  ==>  False
-- ---------------------------------------------------------------------

libre_de_cuadrados :: Integer -> Bool
libre_de_cuadrados n =
    null [x | x <- [2..floor(sqrt(fromInteger(n)))]
            , n `mod` (x*x) == 0]

-- ---------------------------------------------------------------------
-- Ejercicio 11 Definir la función
--    suma_libres_de_cuadrados :: [Int] -> Int
-- tal que (suma_libres_de_cuadrados xs) es la suma de los números
-- libres de cuadrados la lista numérica xs. Por ejemplo,
--    suma_libres_de_cuadrados [6,12,18,30]  ==>  36
-- Nota: Dar dos definiciones, una recursiva y otra por comprensión.
-- ---------------------------------------------------------------------

-- 1ª definición (recursiva):
suma_libres_de_cuadrados_1 :: [Integer] -> Integer
suma_libres_de_cuadrados_1 [] = 0
suma_libres_de_cuadrados_1 (x:xs)
    | libre_de_cuadrados x = x + (suma_libres_de_cuadrados_1 xs)
    | otherwise            = suma_libres_de_cuadrados_1 xs

-- 2ª definición (por comprensión):
suma_libres_de_cuadrados_2 :: [Integer] -> Integer
suma_libres_de_cuadrados_2 xs =
    sum [x | x <- xs, libre_de_cuadrados x]

-- ---------------------------------------------------------------------
-- Ejercicio 12 Definir la función
--    longitud_scm :: [Int] -> [Int] -> Int
-- tal que (longitud_scm xs ys) es la longitud de las subsucesiones
-- comunes maximales de las listas xs e ys. Por ejemplo,
```

```
--    longitud_scm [2,1,4,5,2,3,5,2,4,3] [1,7,5,3,2]  ==>  4
-- ya que [1,5,3,2] es una subsucesión de las dos listas y no poseen
-- ninguna otra subsucesión común de mayor longitud. Obsérvese que los
-- elementos de la subsucesión no son necesariamente elementos adyacentes
-- en las listas.
-- ----------------------------------------------------------------------

longitud_scm :: [Int] -> [Int] -> Int
longitud_scm [] _ = 0
longitud_scm _ [] = 0
longitud_scm (x:xs) (y:ys)
    | x==y = 1 + longitud_scm xs ys
    | otherwise = max (longitud_scm (x:xs) ys) (longitud_scm xs (y:ys))
```

Capítulo 10

Práctica 4b (7 de enero de 2008)

10.1. Práctica 4b en Prolog

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 1: Definir el predicado divisores(+N,-L) que se verifique si
% L es la lista de los divisores del número N. Por ejemplo,
%    ?- divisores(24,L).
%    L = [1, 2, 3, 4, 6, 8, 12, 24]
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

divisores(N,L) :-
   findall(X,((between(1,N,X), N mod X =:= 0)),L).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 2: Definir la propiedad primo(+N) que se verifique si N es
% un número primo. Por ejemplo,
%    ?- primo(7).
%    Yes
%    ?- primo(9).
%    No
% [Indicación: Hacer distintas definiciones basadas en las siguientes
% ideas:
% (1) X es primo si el número de divisores de X es menor que 3 [Usar la
%     definición anterior de divisores].
% (2) X es primo si no hay ningún número entre 2 y la raiz cuadrada de X
%     que sea un divisor de X.
% (3) X es primo si no hay ningún número entre 2 y la raiz cuadrada de X
%     que sea primo y divida a X.]
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% primo(+X)
%    se verifica si X es primo.
primo(X) :-
   primo_1(X).

% 1ª definición:
primo_1(X) :-
   divisores(X,L),
   length(L,N),
   N < 3.

% 2ª definición:
primo_2(X) :-
   Y is floor(sqrt(X)),
   not((between(2,Y,Z), X mod Z =:= 0)).

% 3ª definición:
primo_3(X) :-
   Y is floor(sqrt(X)),
   not((between(2,Y,Z), primo_3(Z), X mod Z =:= 0)).

% ?- time(primo_1(11119)).
% % 11,137 inferences, 0.02 CPU in 0.01 seconds (138% CPU, 556850 Lips)
% Yes
% ?- time(primo_2(11119)).
% % 108 inferences, 0.00 CPU in 0.00 seconds (0% CPU, Infinite Lips)
% Yes
% ?- time(primo_3(11119)).
% % 2,232 inferences, 0.00 CPU in 0.00 seconds (0% CPU, Infinite Lips)
% Yes

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 3: Se considera el siguiente polinomio p(X)=X^2-X+41.
% Definir la relación genera_primo(+X,+Y,-L) que se verifique si L es el
% conjunto de los pares (A,p(A)) tales A es un número del intervalo
% [X,Y] y p(A) es primo. Por ejemplo,
%    ?- genera_primo(5,7,L).
%    L = [[5, 61], [6, 71], [7, 83]]
% Comprobar que p(A) es primo para A desde 1 hasta 40, pero no es primo
```

```
% para A=41.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

genera_primo(X,Y,L) :-
   findall([A,B],((between(X,Y,A), B is A^2-A+41, primo(B))),L).

% Comprobaciones:
%    ?- genera_primo(1,40,_L), length(_L,40).
%    Yes
%    ?- genera_primo(41,41,L).
%    L = []

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 4: Se dice que dos números X e Y son amigos cuando X es
% igual a la suma de los divisores de Y (exceptuando al propio Y) y
% viceversa. Por ejemplo, 6 es amigo de sí mismo puesto que los
% divisores de 6 (exceptuando al 6) son 1, 2 y 3 y 6=1+2+3. Igualmente
% 28 es amigo de sí mismo, puesto que 28=1+2+4+7+14. Definir el
% predicado amigos(+A,+B,-L) que se verifique si L es la lista de las
% parejas de números amigos comprendidos entre A y B. Por ejemplo,
%    ?- amigos(1,300,L).
%    L = [6-6, 28-28, 220-284, 284-220]
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

amigos(A,B,L) :-
   findall(X-Y, ((between(A,B,X),
                  divisores_propios(X,Dx),
                  sumlist(Dx,Y),
                  between(A,B,Y),
                  divisores_propios(Y,Dy),
                  sumlist(Dy,X))),
           L).

divisores_propios(X,L) :-
   X1 is X-1,
   findall(N,((between(1,X1,N), X mod N =:= 0)),L).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 5: Definir la relación lista_a_conjunto(+L,-C) que se
% verifique si C es el conjunto de los elementos de la lista no vacía
```

```
% L. Por ejemplo,
%    ?- lista_a_conjunto([b,c,d,a,c,f,a],C).
%    C = [a, b, c, d, f]
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

lista_a_conjunto(L,C) :-
   setof(X,member(X,L),C).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 6: Definir la relación para_todos(+P,+L) que se verifica si
% todos los elementos de la lista L cumplen la propiedad P. Por ejemplo,
%    ?- para_todos(number,[1,2,3]).
%    Yes
%    ?- para_todos(number,[1,2,3,a]).
%    No
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

para_todos(_P,[]).
para_todos(P,[X|L]) :-
   apply(P,[X]),
   para_todos(P,L).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 7: Definir la relación existe(+P,+L) que se verifica si
% existe algún elemento de la lista L que cumple la propiedad P. Por
% ejemplo,
%    ?- existe(number,[a,b,c]).
%    No
%    ?- existe(number,[a,1,b,c]).
%    Yes
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

existe(P,[X|_]) :-
   apply(P,[X]), !.
existe(P,[_|L]) :-
   existe(P,L).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 8: Definir la relación sublista(+P,+L1,?L2) que se verifica
% si L2 es la lista de los elementos de L1 que verifican la propiedad
```

```
% P. Por ejemplo,
%    ?- sublista(number,[1,a,2,b,1],L).
%    L = [1, 2, 1]
% (Nota: sublista/3 corresponde a la relación definida sublist/3).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

sublista(P,L1,L2) :-
   sublista_1(P,L1,L2).

% 1ª defininición (con negación):
sublista_1(_P,[],[]).
sublista_1(P,[X|L1],[X|L2]) :-
   Q =.. [P,X],
   Q,
   sublista_1(P,L1,L2).
sublista_1(P,[X|L1],L2) :-
   Q =.. [P,X],
   \+ Q,
   sublista_1(P,L1,L2).

% 2ª definición (con corte):
sublista_2(_P,[],[]).
sublista_2(P,[X|L1],[X|L2]) :-
   Q =.. [P,X],
   Q, !,
   sublista_1(P,L1,L2).
sublista_2(P,[_X|L1],L2) :-
   % Q =.. [P,_X],
   % \+ Q,
   sublista_2(P,L1,L2).

% 3ª definición (con findall):
sublista_3(P,L1,L2) :-
   findall(X,(member(X,L1),apply(P,[X])),L2).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 9: [Feb-2000] Definir el predicado rotaciones(L1,L2) que se
% verifique si L2 es la lista cuyos elementos son las listas formadas
% por las sucesivas rotaciones de los elementos de la lista L1. Por
% ejemplo,
```

```
%    ?- rotaciones([1,2,3,4],L).
%    L = [[1, 2, 3, 4], [2, 3, 4, 1], [3, 4, 1, 2], [4, 1, 2, 3]]
%    ?- rotaciones([2,3,4,1],L).
%    L = [[2, 3, 4, 1], [3, 4, 1, 2], [4, 1, 2, 3], [1, 2, 3, 4]]
%    No
% (NOTA: No importa el orden de los elementos en L2.)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

rotaciones(L1,L2) :-
   rotaciones_1(L1,L2).

% 1ª definición:
rotaciones_1(L1,L2) :-
   findall(L,es_rotacion(L1,L),L2).

% Ejemplo para es_rotacion/2:
%  L1 = [1,2,3,4]
%  append(L3,[X|L4],L1)  =>  L3=[], X=1, L4=[2,3,4]
%  append([X|L4],L3,L2)  =>  L2=[1,2,3,4] ;
%  append(L3,[X|L4],L1)  =>  L3=[1], X=2, L4=[3,4]
%  append([X|L4],L3,L2)  =>  L2=[2,3,4,1] ;
%  append(L3,[X|L4],L1)  =>  L3=[1,2], X=3, L4=[4]
%  append([X|L4],L3,L2)  =>  L2=[3,4,1,2] ;
%  append(L3,[X|L4],L1)  =>  L3=[1,2,3], X=4, L4=[]
%  append([X|L4],L3,L2)  =>  L2=[4,1,2,3] ;
es_rotacion(L1,L2)  :-
   append(L3,[X|L4],L1),
   append([X|L4],L3,L2).

% 2ª definición:
rotaciones_2(L1,L2) :-
   findall(L,
           (append(L3,[X|L4],L1),append([X|L4],L3,L)),
           L2).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 10: [Ex. Feb. 2000] Definir la relación capicúas(N,L) que
% tome como dato de entrada el número entero positivo N y devuelva  la
% lista L formada por todos los números enteros positivos capicúa
% menores que N. Por ejemplo,
```

```
%    ?- capicúas(100,L).
%    L = [1,2,3,4,5,6,7,8,9,11,22,33,44,55,66,77,88,99] ;
%    No
%    ?- capicúas(1000,L).
%    L = [1,2,3,4,5,6,7,8,9,11,22,33,44,55,...,979,989,999] ;
%    No
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

capicúas(N,L) :-
   setof(X,(between(1,N,X),es_capicúa(X)),L).

es_capicúa(N) :-
   name(N,L),
   reverse(L,L).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 11: Definir la relación invierte_palabra(P1,P2) que tome
% como dato de entrada una palabra P1 y devuelva la palabra P2 formada
% invirtiendo el orden de las letras. Por ejemplo,
%    ?- invierte_palabra(arroz,P).
%    P = zorra
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

invierte_palabra(P1,P2) :-
    name(P1,L1),
    reverse(L1,L2),
    name(P2,L2).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 12: Definir la relación lista_mayor(+L1,-L2) que se verifica
% si L2 es una lista de la lista de listas L1 de máxima longitud. Por
% ejemplo,
%    ?- lista_mayor([[a,b,c],[d,e],[1,2,3]],L).
%    L = [a, b, c] ;
%    L = [1, 2, 3] ;
%    No
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% lista_mayor(+L1,-L2)
%    se verifica si L2 es una lista de la lista de listas L1 de máxima
```

```
%    longitud.
lista_mayor(L1,L2) :-
   lista_mayor_1(L1,L2).

% 1ª solución de lista_mayor/2.
lista_mayor_1(L1,L2) :-
   member(L2,L1),
   length(L2,N2),
   \+ (member(L,L1), length(L,N), N2 < N).

% 2ª definición de lista_mayor/2
lista_mayor_4(L1,L2) :-
   máxima_longitud(L1,N),
   length(L2,N),
   member(L2,L1).

% máxima_longitud(+L,-N)
%    se verifica si N es la máxima longitud de la lista L. Por ejemplo,
%       ?- máxima_longitud([[a],[a,b,c],[b,a]],N).
%       N = 3
máxima_longitud([L],N) :-
   length(L,N).
máxima_longitud([L|R],N) :-
   length(L,N1),
   máxima_longitud(R,N2),
   N is max(N1,N2).

% 3ª definición de lista_mayor/2
lista_mayor_3([X|L1],L2):-
   length(X,N), !,
   lista_mayor_3_aux(L1,N-[X],Lista),
   member(L2,Lista).

lista_mayor_3_aux([],_-Lista,Lista).
lista_mayor_3_aux([Y|L1],N-Ac,Lista) :-
   length(Y,M),
   ( M < N ->
     lista_mayor_3_aux(L1,N-Ac,Lista)
   ; M = N ->
     lista_mayor_3_aux(L1,N-[Y|Ac],Lista)
```

```
   ; % M > N ->
     lista_mayor_3_aux(L1,M-[Y],Lista)
   ).

% ejemplo_1(+N,-L)
%    se verifica si L es la lista [[1],[1,2],[1,2,3],...,[1,2,...,N]]. Por
%    ejemplo,
%       ?- ejemplo(4,L).
%       L = [[1], [1, 2], [1, 2, 3], [1, 2, 3, 4]]
ejemplo_1(N,L) :-
   findall(L1,(between(1,N,M),findall(X,between(1,M,X),L1)),L).

/* Comparaciones
   ?- ejemplo_1(800,L), time(lista_mayor_1(L,L1)).
   % 1,607,598 inferences, 4.36 CPU in 4.35 seconds (100% CPU, 368715 Lips)
   ?- ejemplo_1(800,L), time(lista_mayor_2(L,L1)).
   % 4,002 inferences, 0.04 CPU in 0.04 seconds (97% CPU, 100050 Lips)
   ?- ejemplo_1(800,L), time(lista_mayor_3(L,L1)).
   % 5,631 inferences, 0.01 CPU in 0.02 seconds (56% CPU, 563100 Lips)
*/

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 13: Un mapa puede representarse mediante la relación
% mapa(N,L) donde N es el nombre del mapa y L es la lista de los pares
% formados por cada una de las regiones del mapa y la lista de sus
% regiones vecinas. Por ejemplo, los mapas siguientes
%       +----------+----------+       +----+-----+-----+----+
%       |    a     |     b    |       | a  |  b  |  c  | d  |
%       +----+-----+-----+----+       +----+-----+-----+----+
%       |    |           |    |       | e  |           | f  |
%       | c  |     d     | e  |       +----+     k     +----+
%       |    |           |    |       | g  |           | h  |
%       +----+-----+-----+----+       +----+-----+-----+----+
%       |    f     |     g    |       |    i     |     j    |
%       +----------+----------+       +----------+----------+
% se pueden representar por
%    mapa(ejemplo_1,
%         [a-[b,c,d], b-[a,d,e], c-[a,d,f], d-[a,b,c,e,f,g],
%          e-[b,d,g], f-[c,d,g], g-[d,e,f]]).
%    mapa(ejemplo_2,
```

```
%         [a-[b,e,k],   b-[a,c,e,k], c-[b,d,f,k], d-[c,f,k], e-[a,b,g,k],
%          f-[c,d,h,k], g-[e,i,k],   h-[f,j,k],   i-[g,j,k], j-[i,h,k],
%          k-[a,b,c,d,e,f,g,h,i,j]]).
% Definir la relación coloración(+M,+LC,-S) que se verifica si S es una
% lista de pares formados por una región del mapa M y uno de los colores
% de la lista de colores LC tal que las regiones vecinas tengan colores
% distintos. Por ejemplo,
%    ?- coloración(ejemplo_1,[1,2,3],S).
%    S = [a-1, b-2, c-2, d-3, e-1, f-1, g-2]
% ¿Qué número de colores se necesitan para colorear el segundo
% mapa? ¿De cuántas formas distintas puede colorearse con dicho
% número?
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

mapa(ejemplo_1,
     [a-[b,c,d],
      b-[a,d,e],
      c-[a,d,f],
      d-[a,b,c,e,f,g],
      e-[b,d,g],
      f-[c,d,g],
      g-[d,e,f]]).

mapa(ejemplo_2,
     [a-[b,e,k],
      b-[a,c,e,k],
      c-[b,d,f,k],
      d-[c,f,k],
      e-[a,b,g,k],
      f-[c,d,h,k],
      g-[e,i,k],
      h-[f,j,k],
      i-[g,j,k],
      j-[i,h,k],
      k-[a,b,c,d,e,f,g,h,i,j]]).

mapa(andalucía,
     [huelva-[sevilla,cádiz],
      cádiz-[huelva,sevilla,málaga],
      sevilla-[huelva,cádiz,córdoba,málaga],
```

```
      córdoba-[sevilla,málaga,granada,jaén],
      jaén-[córdoba,granada,almería],
      granada-[córdoba,málaga,jaén,almería]]).

% coloración(+M,+LC,-S)
%    se verifica si S es una lista de pares formados por una región del mapa M
%    y uno de los colores de la lista de colores LC tal que las regiones
%    vecinas tengan colores distintos.
coloración(M,LC,S) :-
   coloración_2(M,LC,S).

% 1ª definición de coloración/3 (por generación y prueba):
coloración_1(M,LC,S) :-
   mapa(M,L),
   coloración_1_aux(L,LC,S).

coloración_1_aux([],_,[]).
coloración_1_aux([R-V|L],LC,[R-C|S]) :-
   member(C,LC),
   coloración_1_aux(L,LC,S),
   not((member(R1,V), member(R1-C,S))).

% 2ª definición de coloración/3 (por generación y prueba con acumulador):
coloración_2(M,LC,S) :-
   mapa(M,L),
   coloración_2_aux(L,LC,[],S).

coloración_2_aux([],_,S,S).
coloración_2_aux([R-V|L],LC,A,S) :-
   member(C,LC),
   not((member(R1,V), member(R1-C,A))),
   coloración_2_aux(L,LC,[R-C|A],S).

/* Comparación
  ?- time(coloración_1(ejemplo_2,[1,2,3,4],S)).
  % 16,705,282 inferences in 22.61 seconds (738845 Lips)
  S = [a-1, b-2, c-1, d-2, e-3, f-3, g-1, h-1, i-2, j-3, k-4]
  Yes

  ?- time(coloración_2(ejemplo_2,[1,2,3,4],S)).
```

```
  % 546 inferences in 0.00 seconds (Infinite Lips)
  S = [k-4, j-3, i-2, h-1, g-1, f-3, e-3, d-2, c-1, b-2, a-1]
  Yes

  ?- findall(_S,coloración_2(ejemplo_2,[1,2,3,4],_S),_L), length(_L,N).
  N = 1032
*/

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 14:  Definir, usando un acumulador, las siguientes relaciones
% * factorial(+X,-Y) que se verifica si Y es el factorial de X,
% * longitud(L,N) que se verifica si N es la longitud de la lista L.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

factorial(X,Y) :-
   factorial_aux(X,1,Y).

factorial_aux(0,Y,Y).
factorial_aux(X,A,Y) :-
   X > 0,
   A1 is X*A,
   X1 is X-1,
   factorial_aux(X1,A1,Y).

longitud(L,N) :-
   longitud_aux(L,0,N).

longitud_aux([],N,N).
longitud_aux([_|L],A,N) :-
   A1 is A+1,
   longitud_aux(L,A1,N).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 15: Consideremos la siguiente versión del problema de las
% torres de Hanoi:
%   * Existen tres postes que llamaremos A, B y C.
%   * Hay N discos en el poste A ordenados por tamaño (el de arriba es
%     el 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.
% Definir la relación hanoi(+N,+A,+B,+C,-L) que se verifica si L es la
% lista de movimientos para mover N discos desde el poste A al poste B
% usando el C como auxiliar. Por ejemplo,
%    ?- hanoi_1(4,a,b,c,L).
%    L = [a-c,a-b,c-b,a-c,b-a,b-c,a-c, a-b, c-b,c-a,b-a,c-b,a-c,a-b,c-b]
%    ?- hanoi_1(3,a,c,b,L).
%    L = [a-c,a-b,c-b,a-c,b-a,b-c,a-c]
%    ?- hanoi_1(3,c,b,a,L).
%    L = [c-b,c-a,b-a,c-b,a-c,a-b,c-b]
% Dar dos versiones (una sin memoria y otra con memoria) y comparar los
% resultados.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

hanoi(N,A,B,C,L) :-
   hanoi_2(N,A,B,C,L).

hanoi_1(1,A,B,_C,[A-B]).
hanoi_1(N,A,B,C,L) :-
   N > 1,
   N1 is N-1,
   hanoi_1(N1,A,C,B,L1),
   hanoi_1(N1,C,B,A,L2),
   append(L1,[A-B|L2],L).

:- dynamic hanoi_2/5.

hanoi_2(1,A,B,_C,[A-B]).
hanoi_2(N,A,B,C,L) :-
   N > 1,
   N1 is N-1,
   lema(hanoi_2(N1,A,C,B,L1)),
   hanoi_2(N1,C,B,A,L2),
   append(L1,[A-B|L2],L).

lema(P) :-
   P,
   asserta((P :- !)).
```

```
/* Comparación:
  ?- time(hanoi_1(19,a,b,c,_L)).
  % 5,767,165 inferences in 38.51 seconds (149758 Lips)
  ?- time(hanoi_2(19,a,b,c,_L)).
  % 983,429 inferences in 8.62 seconds (114087 Lips)
*/

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 16: El problema de la bandera tricolor consiste en lo
% siguiente: Dada un lista de objetos L1, que pueden ser rojos,
% amarillos o morados, se pide devolver una lista L2 que contiene los
% elementos de L1, primero los rojos, luego los amarillos y por último
% los morados. Definir la relación bandera_tricolor(+L1,-L2) que se
% verifica si L2 es la solución del problema de la bandera tricolor
% correspondiente a la lista L1. Por ejemplo,
%    ?- bandera_tricolor([m,m,a,a,r,r],L).
%    L = [r, r, a, a, m, m]
% Se dispone de los hechos
%    rojo(r).
%    amarillo(a).
%    morado(m).
% Dar distintas definiciones y comparar su eficiencia.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

rojo(r).
amarillo(a).
morado(m).

% (1) Definición de rojo_amarillo_morado(L).
% ----------------------------------------

bandera_tricolor(L1,L2) :-
   permutación(L1,L2),
   rojo_amarillo_morado(L2).

permutación([],[]).
permutación(L1,[X|L2]) :-
   select(X,L1,L3),
   permutación(L3,L2).
```

```
rojo_amarillo_morado(L) :-
   append(Rojo,Amarillo_y_morado,L),
   lista(rojo,Rojo),
   append(Amarillo,Morado,Amarillo_y_morado),
   lista(amarillo,Amarillo),
   lista(morado,Morado).

lista(_,[]).
lista(C,[X|L]) :-
   apply(C,[X]),
   lista(C,L).

% (2) Definición con separación mediante bagof
% --------------------------------------------

bandera_tricolor_2(L1,L2) :-
   bagof(X,(member(X,L1),rojo(X)),Rojos),
   bagof(X,(member(X,L1),amarillo(X)),Amarillos),
   bagof(X,(member(X,L1),morado(X)),Morados),
   append(Amarillos,Morados,AmarillosMorados),
   append(Rojos,AmarillosMorados,L2).

% (3a) Definición mediante selección y negación
% ---------------------------------------------

bandera_tricolor_3a(L1,L2) :-
   bandera_tricolor_3a_aux(L1,[],L2).

bandera_tricolor_3a_aux(L1,L2,L3) :-
   select(X,L1,RL1), morado(X),
   bandera_tricolor_3a_aux(RL1,[X|L2],L3).
bandera_tricolor_3a_aux(L1,L2,L3) :-
   not((select(X,L1,_),morado(X))),
   select(Y,L1,RL1), amarillo(Y),
   bandera_tricolor_3a_aux(RL1,[Y|L2],L3).
bandera_tricolor_3a_aux(L1,L2,L3) :-
   not((select(X,L1,_), morado(X))),
   not((select(Y,L1,_), amarillo(Y))),
   append(L1,L2,L3).
```

```
% (3b) Definición mediante selección y corte
% ----------------------------------------

bandera_tricolor_3b(L1,L2) :-
   bandera_tricolor_3b_aux(L1,[],L2).

bandera_tricolor_3b_aux(L1,L2,L3) :-
   select(X,L1,RL1), morado(X), !,
   bandera_tricolor_3b_aux(RL1,[X|L2],L3).
bandera_tricolor_3b_aux(L1,L2,L3) :-
   % not((select(X,L1,_), morado(X))),
   select(Y,L1,RL1), amarillo(Y), !,
   bandera_tricolor_3b_aux(RL1,[Y|L2],L3).
bandera_tricolor_3b_aux(L1,L2,L3) :-
   % not((select(X,L1,_), morado(X))),
   % not((select(Y,L1,_), amarillo(Y))),
   append(L1,L2,L3).

% (4) Definición mediante separación
% ---------------------------------

bandera_tricolor_4(L1,L2) :-
   separa_2(L1,R,A,M),
   une(R,A,M,L2).

separa_2([],[],[],[]).
separa_2([X|L],[X|R],A,M) :-
   rojo(X), !,
   separa_2(L,R,A,M).
separa_2([X|L],R,[X|A],M) :-
   amarillo(X), !,
   separa_2(L,R,A,M).
separa_2([X|L],R,A,[X|M]) :-
   % morado(X),
   separa_2(L,R,A,M).

une(R,A,M,L) :-
   append(A,M,AM),
   append(R,AM,L).
```

```
% (5) Definición de O'Keefe
% ------------------------

bandera_tricolor_5(L1,L2) :-
   aux_1(L1,L2,X,X,Y,Y,[]).

aux_1([],R,R,B,B,A,A).
aux_1([X|Resto],R0,R,B0,B,A0,A) :-
   color(X,Color),
   aux_2(Color,R0,R,B0,B,A0,A,X,Resto).

aux_2(rojo,[X|R1],R,B0,B,A0,A,X,Resto) :-
   aux_1(Resto,R1,R,B0,B,A0,A).
aux_2(amarillo,R0,R,[X|B1],B,A0,A,X,Resto) :-
   aux_1(Resto,R0,R,B1,B,A0,A).
aux_2(morado,R0,R,B0,B,[X|A1],A,X,Resto) :-
   aux_1(Resto,R0,R,B0,B,A1,A).

color(X,Color) :-
   member(Color,[morado,rojo,amarillo]),
   Atom =.. [Color,X],
   Atom.

% Comparación:
% ------------

% ?- time(bandera_tricolor([m,a,r,m,a,r,m,a,r],L)).
% 13,193,039 inferences, 4,05 CPU in 4,05 seconds (100% CPU, 3257540 Lips)
% L = [r, r, r, b, b, b, a, a, a]
%
% ?- time(bandera_tricolor_2([m,a,r,m,a,r,m,a,r],L)).
% % 151 inferences in 0.00 seconds (Infinite Lips)
% L = [r, r, r, a, a, a, m, m, m]
%
% ?- time(bandera_tricolor_3a([m,a,r,m,a,r,m,a,r],L)).
% % 149 inferences in 0.00 seconds (Infinite Lips)
% L = [r, r, r, a, a, a, m, m, m]
%
% ?- time(bandera_tricolor_3b([m,a,r,m,a,r,m,a,r],L)).
% % 89 inferences in 0.00 seconds (Infinite Lips)
```

```
% L = [r, r, r, a, a, a, m, m, m]
%
% ?- time(bandera_tricolor_4([m,a,r,m,a,r,m,a,r],L)).
% % 35 inferences in 0.00 seconds (Infinite Lips)
% L = [r, r, r, a, a, a, m, m, m]
%
% ?- time(bandera_tricolor_5([m,a,r,m,a,r,m,a,r],L)).
% % 83 inferences in 0.00 seconds (Infinite Lips)
% L = [r, r, r, a, a, a, m, m, m]

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ejercicio 17: Se dice que L es una sucesión de Lanford si L es una
% lista de longitud 27 en la cual aparecen 3 veces cada uno de los
% dígitos del 1 al 9 y que además cumple la propiedad de que entre dos 1
% siempre hay un dígito, entre dos 2 hay  dos dígitos, entre dos 3 hay
% tres dígitos, etc.
% Definir la relación lanford(?L) que se verifique si L es una longitud
% de Lanford. Por ejemplo,
%    ?- lanford(L).
%    L = [1,9,1,2,1,8,2,4,6,2,7,9,4,5,8,6,3,4,7,5,3,9,6,8,3,5,7] ;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% lanford(?L)
%    se verifica si L es una sucesión de Lanford.
lanford(L):-
   length(L,27),
   sublista([1,_,1,_,1],L),
   sublista([2,_,_,2,_,_,2],L),
   sublista([3,_,_,_,3,_,_,_,3],L),
   sublista([4,_,_,_,_,4,_,_,_,_,4],L),
   sublista([5,_,_,_,_,_,5,_,_,_,_,_,5],L),
   sublista([6,_,_,_,_,_,_,6,_,_,_,_,_,_,6],L),
   sublista([7,_,_,_,_,_,_,_,7,_,_,_,_,_,_,_,7],L),
   sublista([8,_,_,_,_,_,_,_,_,8,_,_,_,_,_,_,_,_,8],L),
   sublista([9,_,_,_,_,_,_,_,_,_,9,_,_,_,_,_,_,_,_,_,9],L).

sublista(L1,L2):-
   append(_,L3,L2),
   append(L1,_,L3).
```

```
% Experimentos:
%    ?- time(lanford(L)).
%    % 48,547 inferences, 0.04 CPU in 0.03 seconds (115% CPU, 1213675 Lips)
%    L = [1,9,1,2,1,8,2,4,6,2,7,9,4,5,8,6,3,4,7,5,3,9,6,8,3,5,7] ;
%    No
%
%    ?- time((findall(S,lanford(S),_L),length(_L,N))).
%    % 7,130,914 inferences, 2.56 CPU in 2.56 seconds (100% CPU, 2785513 Lips)
%    N = 6
%    Yes
```

10.2. Práctica 4b en Haskell

```
-- ---------------------------------------------------------------------
-- Librerías auxiliares
-- ---------------------------------------------------------------------

import Data.List
default (Integer,Float,Double)

-- ---------------------------------------------------------------------
-- Ejercicio 1: Definir la función
--    divisores ::Integer -> [Integer]
-- tal que (divisores n) es la lista de los divisores del número n. Por
-- ejemplo,
--    divisores 24  ==>  [1,2,3,4,6,8,12,24]
-- ---------------------------------------------------------------------

divisores ::Integer -> [Integer]
divisores n =
    [x | x <- [1..n], n `mod` x == 0]

-- ---------------------------------------------------------------------
-- Ejercicio 2: Definir la función
--    primo :: Integer -> Bool
-- tal que (primo n) se verifica si n es un número primo. Por ejemplo,
--    primo_1 7  ==>  True
--    primo_1 9  ==>  False
-- [Nota: Dar dos definiciones basadas en las siguientes ideas:
-- (1) X es primo si el número de divisores de X es menor que 3 [Usar la
```

```
--     definición anterior de divisores].
-- (2) X es primo si no hay ningún número entre 2 y la raiz cuadrada de X
--     que sea un divisor de X.]
-- ---------------------------------------------------------------------

-- 1ª definición:
primo_1 :: Integer -> Bool
primo_1 x =
    length (divisores x) < 3

-- 2ª definición:
primo_2 :: Integer -> Bool
primo_2 x =
    null [y | y <- [2..floor(sqrt(fromInteger(x)))]
            , x `mod` y == 0]

-- Definición de primo:
primo :: Integer -> Bool
primo = primo_1

-- ---------------------------------------------------------------------
-- Ejercicio 3: Se considera el siguiente polinomio p(X)=X^2-X+41.
-- Definir la función
--    genera_primo :: Integer -> Integer -> [(Integer,Integer)]
-- tal que (genera_primo x y) es el conjunto de los pares (a,p(a)) tales
-- a es un número del intervalo [x,y] y p(a) es primo. Por ejemplo,
--    genera_primo 5 7  ==>  [(5,61),(6,71),(7,83)]
-- Comprobar que p(a) es primo para a desde 1 hasta 40, pero no es primo
-- para a=41.
-- ---------------------------------------------------------------------

-- La definición es
genera_primo :: Integer -> Integer -> [(Integer,Integer)]
genera_primo x y =
    [(a,b) | a <- [x..y], let b = a^2-a+41, primo b]

-- La comprobación es
--    Main> [x | (x,y) <- genera_primo 1 41]
--    [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,
--     22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40]
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 4: Se dice que dos números X e Y son amigos cuando X es
-- igual a la suma de los divisores de Y (exceptuando al propio Y) y
-- viceversa. Por ejemplo, 6 es amigo de sí mismo puesto que los
-- divisores de 6 (exceptuando al 6) son 1, 2 y 3 y 6=1+2+3. Igualmente
-- 28 es amigo de sí mismo, puesto que 28=1+2+4+7+14. Definir la función
--    amigos :: Int -> Int -> [(Int,Int)]
-- tal que (amigos a b) es la lista de las parejas de números amigos
-- comprendidos entre a y b. Por ejemplo,
--    amigos 1 300  ==>  [(6,6),(28,28),(220,284),(284,220)]
-- ---------------------------------------------------------------------

amigos :: Int -> Int -> [(Int,Int)]
amigos a b =
    [(x,y) |
     x <- [a..b],
     let y = sum (divisores_propios x),
     a <= y, y <= b,
     sum (divisores_propios y) == x]

divisores_propios :: Int -> [Int]
divisores_propios x =
    [n | n <- [1..x-1], x `mod` n == 0]

-- ---------------------------------------------------------------------
-- Ejercicio 5 Definir la función
--    lista_a_conjunto :: Eq a => [a] -> [a]
-- tal que (lista_a_conjunto xs) es el conjunto correspondiente a la
-- lista xs (es decir, contiene los mismos elementos que xs en el mismo
-- orden, pero si xs tiene elementos repetidos sólo se incluye la última
-- aparición de cada elemento). Por ejemplo,
--    lista_a_conjunto [2,1,2,0]  ==> [1,2,0]
-- Nota: La función lista_a_conjunto se corresponde con la función nub
-- definida en la librería Data.List.
-- ---------------------------------------------------------------------

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

```
    | elem x xs = lista_a_conjunto xs
    | otherwise = x:(lista_a_conjunto xs)

-- ---------------------------------------------------------------------
-- Ejercicio 6: Definir la función
--    para_todos :: (a -> Bool) -> [a] -> Bool
-- tal que (para_todos p xs) se verifica si todos los elementos de la
-- lista xs cumplen la propiedad p. Por ejemplo,
--    para_todos even [2,4,6]    ==>  True
--    para_todos even [2,4,7,6]  ==>  False
-- Dar una definición recursiva y otra por comprensión.
-- ---------------------------------------------------------------------

-- 1ª definición (por recursión):
para_todos :: (a -> Bool) -> [a] -> Bool
para_todos _ []     = True
para_todos p (x:xs) = (p x) && (para_todos p xs)

-- 2ª definición (por comprensión):
para_todos' :: (a -> Bool) -> [a] -> Bool
para_todos' p xs = and [p x | x <- xs]

-- ---------------------------------------------------------------------
-- Ejercicio 7: Definir la función
--    existe :: (a -> Bool) -> [a] -> Bool
-- tal que (existe p xs) se verifica si existe algún elemento de la
-- lista xs que cumple la propiedad p. Por ejemplo,
--    existe odd [2,4,6]    ==>  False
--    existe odd [2,4,7,6]  ==>  True
-- Dar una definición recursiva y otra por comprensión.
-- ---------------------------------------------------------------------

-- 1ª definición (por recursión):
existe :: (a -> Bool) -> [a] -> Bool
existe _ []     = False
existe p (x:xs) = (p x) || (existe p xs)

-- 2ª definición (por comprensión):
existe' :: (a -> Bool) -> [a] -> Bool
existe' p xs = or [p x | x <- xs]
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 8: Definir la relación sublista(+P,+L1,?L2) que se verifica
-- si L2 es la lista de los elementos de L1 que verifican la propiedad
-- P. Por ejemplo,
--    sublista even [1,3,2,5,6,8]  ==>  [2,6,8]
-- Dar una definición recursiva y otra por comprensión.
-- ---------------------------------------------------------------------

-- 1ª definición (recursiva):
sublista :: (a -> Bool) -> [a] -> [a]
sublista _ [] = []
sublista p (x:xs)
    | p x       = x:(sublista p xs)
    | otherwise = sublista p xs

-- 2ª definición (por comprensión):
sublista' :: (a -> Bool) -> [a] -> [a]
sublista' p xs = [x | x <-xs, p x]

-- ---------------------------------------------------------------------
-- Ejercicio 9 Definir la función
--    rotaciones :: [a]-> [[a]]
-- tal que (rotaciones xs) es la lista cuyos elementos son las listas
-- formadas por las sucesivas rotaciones de los elementos de la lista
-- xs. Por ejemplo,
--    rotaciones [1,2,3,4] ==> [[1,2,3,4],[2,3,4,1],[3,4,1,2],[4,1,2,3]]
-- ---------------------------------------------------------------------

rotaciones :: [a]-> [[a]]
rotaciones xs =
    [bs++as | n <- [0..(length xs)-1], let (as,bs) = splitAt n xs]
-- ---------------------------------------------------------------------
-- Ejercicio 10 Definir la función
--    capicúas :: Int -> [Int]
-- tal que (capicúas n) es la lista formada por todos los números
-- enteros positivos capicúa menores que n. Por ejemplo,
--    capicúas 100  ==>  [1,2,3,4,5,6,7,8,9,11,22,33,44,55,66,77,88,99]
--    capicúas 1000 ==>  [1,2,3,4,5,6,7,8,9,11,22,33,...,979,989,999]
-- ---------------------------------------------------------------------
```

```
capicúas :: Int -> [Int]
capicúas n =
    [x | x <- [1..n], es_capicúa x]

es_capicúa :: Int -> Bool
es_capicúa n =
    reverse c == c
    where c = show n

-- ---------------------------------------------------------------------
-- Ejercicio 11: Definir la función
--    invierte_palabra :: String -> String
-- tal que ( invierte_palabra p) es la palabra formada invirtiendo el
-- orden de las letras de la palabra p. Por ejemplo,
--    invierte_palabra "arroz"  ==>  "zorra"
-- ---------------------------------------------------------------------

invierte_palabra :: String -> String
invierte_palabra p =
    reverse p

-- ---------------------------------------------------------------------
-- Ejercicio 12: Definir la función
--    listas_mayores :: [[a]] -> [[a]]
-- tal que (listas_mayores xss) es la lista de las listas de xss de
-- máxima longitud. Por ejemplo,
--    listas_mayores [[4,2,7],[1,9],[1,2,3]]  ==>  [[4,2,7],[1,2,3]]
-- ---------------------------------------------------------------------

-- 1ª definición:
listas_mayores :: [[a]] -> [[a]]
listas_mayores xss =
    [xs | xs <- xss, length xs == n]
    where n = maximum [length xs | xs <- xss]

-- 2ª definición:
listas_mayores' :: [[a]] -> [[a]]
listas_mayores' (xs:xss) =
    reverse (listas_mayores'_aux xss (length xs) [xs])
```

```
listas_mayores'_aux :: [[a]] -> Int -> [[a]] -> [[a]]
listas_mayores'_aux [] _ yss = yss
listas_mayores'_aux (xs:xss) n yss
    | length xs <  n = listas_mayores'_aux xss n yss
    | length xs == n = listas_mayores'_aux xss n (xs:yss)
    | otherwise      = listas_mayores'_aux xss (length xs) [xs]

-- ---------------------------------------------------------------------
-- Ejercicio 13: Un mapa puede representarse mediante la lista de los
-- pares formados por cada una de las regiones del mapa y la lista de
-- sus regiones vecinas. Por ejemplo, los mapas siguientes
--       +----------+----------+       +----+-----+-----+----+
--       |    a     |     b    |       | a  |  b  |  c  | d  |
--       +----+-----+-----+----+       +----+-----+-----+----+
--       |    |           |    |       | e  |           | f  |
--       | c  |     d     | e  |       +----+     k     +----+
--       |    |           |    |       | g  |           | h  |
--       +----+-----+-----+----+       +----+-----+-----+----+
--       |    f     |     g    |       |    i     |     j    |
--       +----------+----------+       +----------+----------+
-- se pueden representar por
--    ejemplo_1 =
--        [('a',['b','c','d']),
--         ('b',['a','d','e']),
--         ('c',['a','d','f']),
--         ('d',['a','b','c','e','f','g']),
--         ('e',['b','d','g']),
--         ('f',['c','d','g']),
--         ('g',['d','e','f'])]
--    ejemplo_2 =
--         [('a',"bek"),
--          ('b',"acek"),
--          ('c',"bdfk"),
--          ('d',"cfk"),
--          ('e',"abgk"),
--          ('f',"cdhk"),
--          ('g',"eik"),
--          ('h',"fjk"),
--          ('i',"gjk"),
```

```
--          ('j',"ihk"),
--          ('k',"abcdefghij")]
-- donde se están usando los siguientes tipos
--    type Región    = Char
--    type Mapa      = [(Región,[Región])]
--    type Color     = Int
--    type Coloreado = [(Región,Color)]
-- Definir la función
--    oloraciones :: Mapa -> [Color] -> [Coloreado]
-- tal que (coloaraciones m cs) ee la lista de listas de pares formados
-- por una región del mapa m y uno de los colores de la lista de colores
-- cs tal que las regiones vecinas tengan colores distintos. Por ejemplo,
--    Main> coloraciones ejemplo_1 [1,2,3]
--    [[('a',1),('b',2),('c',2),('d',3),('e',1),('f',1),('g',2)],
--     [('a',1),('b',3),('c',3),('d',2),('e',1),('f',1),('g',3)],
--     [('a',2),('b',1),('c',1),('d',3),('e',2),('f',2),('g',1)],
--     [('a',2),('b',3),('c',3),('d',1),('e',2),('f',2),('g',3)],
--     [('a',3),('b',1),('c',1),('d',2),('e',3),('f',3),('g',1)],
--     [('a',3),('b',2),('c',2),('d',1),('e',3),('f',3),('g',2)]]
-- ¿Qué número de colores se necesitan para colorear el segundo mapa?
-- ¿De cuántas formas distintas puede colorearse con dicho número?
-- ---------------------------------------------------------------------

-- Declaraciones de tipos:
type Región    = Char
type Mapa      = [(Región,[Región])]
type Color     = Int
type Coloreado = [(Región,Color)]

-- Ejemplos de mapas:
ejemplo_1, ejemplo_2 :: Mapa
ejemplo_1 =
    [('a',['b','c','d']),
     ('b',['a','d','e']),
     ('c',['a','d','f']),
     ('d',['a','b','c','e','f','g']),
     ('e',['b','d','g']),
     ('f',['c','d','g']),
     ('g',['d','e','f'])]
ejemplo_2 =
```

```
     [('a',"bek"),
      ('b',"acek"),
      ('c',"bdfk"),
      ('d',"cfk"),
      ('e',"abgk"),
      ('f',"cdhk"),
      ('g',"eik"),
      ('h',"fjk"),
      ('i',"gjk"),
      ('j',"ihk"),
      ('k',"abcdefghij")]

-- 1ª definición (sin acumulador):
coloraciones_1 :: Mapa -> [Color] -> [Coloreado]
coloraciones_1 [] _ = [[]]
coloraciones_1 ((r,v):m) cs =
    [(r,c):mc |
     c <- cs,
     mc <- coloraciones_1 m cs,
     null [r1 | r1 <- v, elem (r1,c) mc]]

-- 2ª definición (con acumulador):
coloraciones_2 :: Mapa -> [Color] -> [Coloreado]
coloraciones_2 m cs =
    coloraciones_2_aux m cs []

coloraciones_2_aux :: Mapa -> [Color] -> Coloreado -> [Coloreado]
coloraciones_2_aux [] _ mc = [mc]
coloraciones_2_aux ((r,v):m) cs mc =
    concat [coloraciones_2_aux m cs ((r,c):mc) |
            c <- cs,
            null [r1 | r1 <- v, elem (r1,c) mc]]

-- ¿Qué número de colores se necesitan para colorear el segundo mapa?
--    Main> head [n | n <- [1..], coloraciones_2 ejemplo_2 [1..n] /= []]
--    4

-- ¿De cuántas formas distintas puede colorearse con dicho número?
--    Main> length (coloraciones_2 ejemplo_2 [1..4])
--    1032
```

```
-- Comentario: La segunda definición es más eficiente:
--    Main> :set +s
--    Main> head (coloraciones_1 ejemplo_2 [1..4])
--    [('a',1),('b',2),('c',1),('d',2),('e',3),('f',3),
--     ('g',1),('h',1),('i',2),('j',3),('k',4)]
--    (76158543 reductions, 132014974 cells, 135 garbage collections)
--    Main> head (coloraciones_2 ejemplo_2 [1..4])
--    [('k',4),('j',3),('i',2),('h',1),('g',1),
--     ('f',3),('e',3),('d',2),('c',1),('b',2),('a',1)]
--    (3444 reductions, 5383 cells)
--    Main> length (coloraciones_1 ejemplo_2 [1..4])
--    1032
--    (1106908363 reductions, 1918457337 cells, 1971 garbage collections)
--    Main> length (coloraciones_2 ejemplo_2 [1..4])
--    1032
--    (31481445 reductions, 45723161 cells, 47 garbage collections)

-- ---------------------------------------------------------------------
-- Ejercicio 14:  Definir, usando un acumulador, las siguientes funciones
--    factorial :: Integer -> Integer
-- tal que (factorial x) es el factorial de x y
--    longitud :: [a] -> Int
-- tal que (longitud xs) es la longitud de la lista xs.
-- ---------------------------------------------------------------------

-- Definición del factorial:
factorial :: Integer -> Integer
factorial x =
    factorial_aux x 1

factorial_aux :: Integer -> Integer -> Integer
factorial_aux 0 y = y
factorial_aux x y | x>0 = factorial_aux (x-1) (x*y)

-- Definición de longitud:
longitud :: [a] -> Int
longitud xs =
    longitud_aux xs 0
```

```
longitud_aux :: [a] -> Int -> Int
longitud_aux []     n = n
longitud_aux (x:xs) n = longitud_aux xs (n+1)

-- ---------------------------------------------------------------------
-- Ejercicio 15: Consideremos la siguiente versión del problema de las
-- torres de Hanoi:
--   * Existen tres postes que llamaremos A, B y C.
--   * Hay N discos en el poste A ordenados por tamaño (el de arriba es
--     el 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.
-- Definir la función
--    hanoi :: Int -> Disco -> Disco -> Disco -> Movimientos
-- tal que (hanoi n a b c) es la lista de movimientos para mover n
-- discos desde el poste a al poste b usando el c como auxiliar. Por
-- ejemplo,
--    Main> hanoi 4 A B C
--    [(A,C),(A,B),(C,B),(A,C),(B,A),(B,C),(A,C),(A,B),
--     (C,B),(C,A),(B,A),(C,B),(A,C),(A,B),(C,B)]
--    Main> hanoi 3 A C B
--    [(A,C),(A,B),(C,B),(A,C),(B,A),(B,C),(A,C)]
--    Main> hanoi 3 C B A
--    [(C,B),(C,A),(B,A),(C,B),(A,C),(A,B),(C,B)]
-- Se usan los siguientes tipos:
--    data Disco = A | B | C deriving Show
--    type Movimientos = [(Disco,Disco)]
-- ---------------------------------------------------------------------

data Disco = A | B | C deriving Show
type Movimientos = [(Disco,Disco)]

hanoi :: Int -> Disco -> Disco -> Disco -> Movimientos
hanoi 1     a b c = [(a,b)]
hanoi (n+1) a b c = (hanoi n a c b) ++ [(a,b)] ++ (hanoi n c b a)

-- ---------------------------------------------------------------------
-- Ejercicio 16: El problema de la bandera tricolor consiste en lo
```

```
-- siguiente: Dada un lista de objetos L1, que pueden ser rojos,
-- amarillos o morados, se pide devolver una lista L2 que contiene los
-- elementos de L1, primero los rojos, luego los amarillos y por último
-- los morados. Definir la función
--     bandera_tricolor :: [Color'] -> [Color']
-- tal que (bandera_tricolor xs) es la solución del problema de la
-- bandera tricolor correspondiente a la lista xs. Por ejemplo,
--     bandera_tricolor [Mo,Mo,Am,Am,Ro,Ro]  ==>  [Ro,Ro,Am,Am,Mo,Mo]
-- El tipo Color' se define por
--     data Color' = Ro | Am | Mo deriving (Show, Eq)
-- ---------------------------------------------------------------------

data Color' = Ro | Am | Mo deriving (Show, Eq)

bandera_tricolor :: [Color'] -> [Color']
bandera_tricolor xs =
    [x | x <- xs, x == Ro] ++
    [x | x <- xs, x == Am] ++
    [x | x <- xs, x == Mo]

-- ---------------------------------------------------------------------
-- Ejercicio 17: Se dice que L es una sucesión de Lanford si L es una
-- lista de longitud 27 en la cual aparecen 3 veces cada uno de los
-- dígitos del 1 al 9 y que además cumple la propiedad de que entre dos 1
-- siempre hay un dígito, entre dos 2 hay  dos dígitos, entre dos 3 hay
-- tres dígitos, etc.
-- Definir la relación lanford(?L) que se verifique si L es una longitud
-- de Lanford. Por ejemplo,
--    ?- lanford(L).
--    L = [1,9,1,2,1,8,2,4,6,2,7,9,4,5,8,6,3,4,7,5,3,9,6,8,3,5,7] ;
-- ---------------------------------------------------------------------

-- Un estado es triple formado por un número, una lista de 3 números
-- (que representan sus posiciones) y una lista de números (que
-- representan las posiciones libres).
type Estado = (Int,[Int],[Int])

-- Definición de la función lanford:
lanford :: [[Int]]
lanford =
```

```
    [listaSolución s | s <- lanfordAux]

-- lanfordAux es la lista de soluciones representadas como estados. Por
-- ejemplo,
--    Main> head lanfordAux
--    [(9,[2,12,22],[]),
--     (8,[6,15,24],[2,12,22]),
--     (7,[11,19,27],[2,6,12,15,22,24]),
--     (6,[9,16,23],[2,6,11,12,15,19,22,24,27]),
--     (5,[14,20,26],[2,6,9,11,12,15,16,19,22,23,24,27]),
--     (4,[8,13,18],[2,6,9,11,12,14,15,16,19,20,22,23,24,26,27]),
--     (3,[17,21,25],[2,6,8,9,11,12,13,14,15,16,18,19,20,22,23,24,26,27]),
--     (2,[4,7,10],[2,6,8,9,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,
--                  26,27]),
--     (1,[1,3,5],[2,4,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,
--                 24,25,26,27])]
lanfordAux :: [[Estado]]
lanfordAux =
    concat [soluciones e [] | e <- iniciales]

-- (soluciones e vis) es la lista de soluciones a partir del estado e
-- habiendo visitado los estados de la lista vis.
soluciones :: Estado -> [Estado] -> [[Estado]]
-- soluciones e@(a,xs,ys) vis
--     | trace ("==> " ++ show e) False = undefined
soluciones e@(a,xs,ys) vis
    | null ys   = [e:vis]
    | otherwise = concat [soluciones e' (e:vis) | e' <- sucesores e]

-- (sucesores e) es la lista de los sucesores del estado e. Por ejemplo,
--    Main> sucesores (5,[14,20,26],[2,6,9,11,12,15,16,19,22,23,24,27])
--    [(6,[2,9,16],[6,11,12,15,19,22,23,24,27]),
--     (6,[9,16,23],[2,6,11,12,15,19,22,24,27])]
sucesores :: Estado -> [Estado]
sucesores (a,_,xs) =
    [(a+1,posiciones,(xs \\ posiciones)) |
     x <- xs,
     let posiciones = [x,x+a+2,x+2*(a+2)],
     subconjunto posiciones xs]
```

```
-- (subconjunto xs ys) se verifica si xs es subconjunto de ys.
subconjunto []     ys = True
subconjunto (x:xs) ys = (elem x ys) && (subconjunto xs ys)

-- iniciales es la lista de los estados iniciales.
iniciales :: [Estado]
iniciales =
    sucesores (0,[],[1..27])

-- (listaSolución s) es la lista correspondiente a la solución s. Por
-- ejemplo,
--    listaSolución [(1,[2,4],[]),(7,[1,3],[])]  ==>  [7,1,7,1]
listaSolución :: [Estado] -> [Int]
listaSolución s =
    (snd . unzip . sort) [(n,a) | (a,xs,ys) <- s, n <- xs]
```