Exámenes de
“Programación funcional con Haskell”

Vol. 1 (Curso 2009–2010)

José A. Alonso Jiménez

Grupo de Lógica Computacional
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, 17 de diciembre de 2010

Índice general

Introducción **5**

1 Exámenes del grupo 1 **7**

José A. Alonso y Gonzalo Aranda

1.1 Examen 1 (30 de noviembre de 2009) 7
1.2 Examen 2 (12 de febrero de 2010) 9
1.3 Examen 3 (15 de marzo de 2010) 11
1.4 Examen 4 (12 de abril de 2010) 16
1.5 Examen 5 (17 de mayo de 2010) 20
1.6 Examen 6 (21 de junio de 2010) 22
1.7 Examen 7 (5 de julio de 2010) 25
1.8 Examen 8 (15 de septiembre de 2010) 27
1.9 Examen 9 (17 de diciembre de 2010) 32

2 Exámenes del grupo 2 **37**

María J. Hidalgo

2.1 Examen 1 (4 de diciembre de 2009) 37
2.2 Examen 2 (16 de marzo de 2010) 39
2.3 Examen 3 (5 de julio de 2010) 44
2.4 Examen 4 (15 de septiembre de 2010) 44
2.5 Examen 5 (17 de diciembre de 2010) 44

A Resumen de funciones predefinidas de Haskell **45**
A.1 Resumen de funciones sobre TAD en Haskell 47

B Método de Pólya para la resolución de problemas **51**
B.1 Método de Pólya para la resolución de problemas matemáticos . . 51
B.2 Método de Pólya para resolver problemas de programación 52

Bibliografía **55**

Introducción

Este libro es una recopilación de las soluciones de ejercicios de los exámenes de programación funcional con Haskell de la asignatura de Informática del Grado en Matemática de la Universidad de Sevilla.

Los exámenes se realizaron en el aula de informática y su duración fue de 2 horas. La materia de cada examen es la impartida desde el comienzo del curso (generalmente, el 1 de octubre) hasta la fecha del examen. Dicha materia se encuentra en los libros de temas y ejercicios del curso:

- Temas de programación funcional (curso 2009–10) [1]
- Ejercicios de "Informática de 1º de Matemáticas" (2009–10) [2]
- Piensa en Haskell (Ejercicios de programación funcional con Haskell) [3]

El libro consta de 2 capítulos correspondientes a los 2 grupos de la asignatura. En cada capítulo hay una sección por cada uno de los exámenes del grupo. Los ejercicios de cada examen han sido propuestos por los profesores de su grupo (cuyos nombres aparecen en el título del capítulo). Sin embargo, los he modificado para unificar el estilo de su presentación.

Finalmente, el libro contiene dos apéndices. Uno con el método de Polya de resolución de problemas (sobre el que se hace énfasis durante todo el curso) y el otro con un resumen de las funciones de Haskell de uso más frecuente.

Los códigos del libro están disponibles en GitHub [4]

José A. Alonso
Sevilla, 17 de diciembre de 2010

[1] https://www.cs.us.es/~jalonso/cursos/i1m-09/temas/2009-10-IM-temas-PF.pdf
[2] https://www.cs.us.es/~jalonso/cursos/i1m-09/ejercicios/ejercicios-I1M-2009.pdf
[3] http://www.cs.us.es/~jalonso/publicaciones/Piensa_en_Haskell.pdf
[4] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol1

1

Exámenes del grupo 1

José A. Alonso y Gonzalo Aranda

1.1. Examen 1 (30 de noviembre de 2009)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- 1º examen de evaluación continua (30 de noviembre de 2009)
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir, por recursión ,la función
--    sumaFactR :: Int -> Int
-- tal que (sumaFactR n) es la suma de los factoriales de los números
-- desde 0 hasta n. Por ejemplo,
--    sumaFactR 3  ==  10
-- ---------------------------------------------------------------------

sumaFactR :: Int -> Int
sumaFactR 0     = 1
sumaFactR (n+1) = factorial (n+1) + sumaFactR n

-- (factorial n) es el factorial de n. Por ejemplo,
--    factorial 4  ==  24
factorial n = product [1..n]

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir, por comprensión, la función
--    sumaFactC :: Int -> Int
-- tal que (sumaFactC n) es la suma de los factoriales de los números
```

```haskell
-- desde 0 hasta n. Por ejemplo,
--    sumaFactC 3  ==  10
-- ---------------------------------------------------------------------

sumaFactC :: Int -> Int
sumaFactC n = sum [factorial x | x <- [0..n]]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir, por recursión, la función
--    copia :: [a] -> Int -> [a]
-- tal que (copia xs n) es la lista obtenida copiando n veces la lista
-- xs. Por ejemplo,
--    copia "abc" 3  ==  "abcabcabc"
-- ---------------------------------------------------------------------

copia :: [a] -> Int -> [a]
copia xs 0 = []
copia xs n = xs ++ copia xs (n-1)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir, por recursión, la función
--    incidenciasR :: Eq a => a -> [a] -> Int
-- tal que (incidenciasR x ys) es el número de veces que aparece el
-- elemento x en la lista ys. Por ejemplo,
--    incidenciasR 3 [7,3,5,3]  ==  2
-- ---------------------------------------------------------------------

incidenciasR :: Eq a => a -> [a] -> Int
incidenciasR _ []                   = 0
incidenciasR x (y:ys) | x == y      = 1 + incidenciasR x ys
                      | otherwise = incidenciasR x ys

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir, por comprensión, la función
--    incidenciasC :: Eq a => a -> [a] -> Int
-- tal que (incidenciasC x ys) es el número de veces que aparece el
-- elemento x en la lista ys. Por ejemplo,
--    incidenciasC 3 [7,3,5,3]  ==  2
-- ---------------------------------------------------------------------
```

```haskell
incidenciasC :: Eq a => a -> [a] -> Int
incidenciasC x ys = length [y | y <- ys, y == x]
```

1.2. Examen 2 (12 de febrero de 2010)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- 2º examen de evaluación continua (12 de febrero de 2010)
-- ---------------------------------------------------------------------

import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir, por recursión, la función
--    diferenciasR :: Num a => [a] -> [a]
-- tal que (diferenciasR xs) es la lista de las diferencias entre los
-- elementos consecutivos de xs. Por ejemplo,
--    diferenciasR [5,3,8,7]  == [2,-5,1]
-- ---------------------------------------------------------------------

diferenciasR :: Num a => [a] -> [a]
diferenciasR []         = []
diferenciasR [_]        = []
diferenciasR (x1:x2:xs) = (x1-x2) : diferenciasR (x2:xs)

-- La definición anterior puede simplificarse
diferenciasR' :: Num a => [a] -> [a]
diferenciasR' (x1:x2:xs) = (x1-x2) : diferenciasR' (x2:xs)
diferenciasR' _          = []

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir, por comprensión, la función
--    diferenciasC :: Num a => [a] -> [a]
-- tal que (diferenciasC xs) es la lista de las diferencias entre los
-- elementos consecutivos de xs. Por ejemplo,
--    diferenciasC [5,3,8,7]  == [2,-5,1]
-- ---------------------------------------------------------------------

diferenciasC :: Num a => [a] -> [a]
diferenciasC xs = [a-b | (a,b) <- zip xs (tail xs)]
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    producto :: [[a]] -> [[a]]
-- tal que (producto xss) es el producto cartesiano de los conjuntos
-- xss. Por ejemplo,
--    ghci> producto [[1,3],[2,5]]
--    [[1,2],[1,5],[3,2],[3,5]]
--    ghci> producto [[1,3],[2,5],[6,4]]
--    [[1,2,6],[1,2,4],[1,5,6],[1,5,4],[3,2,6],[3,2,4],[3,5,6],[3,5,4]]
--    ghci> producto [[1,3,5],[2,4]]
--    [[1,2],[1,4],[3,2],[3,4],[5,2],[5,4]]
--    ghci> producto []
--    [[]]
-- ---------------------------------------------------------------------

producto :: [[a]] -> [[a]]
producto []       = [[]]
producto (xs:xss) = [x:ys | x <- xs, ys <- producto xss]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir el predicado
--    comprueba :: [[Int]] -> Bool
-- tal que tal que (comprueba xss) se verifica si cada elemento de la
-- lista de listas xss contiene algún número par. Por ejemplo,
--    comprueba [[1,2],[3,4,5],[8]]  ==  True
--    comprueba [[1,2],[3,5]]        ==  False
-- ---------------------------------------------------------------------

-- 1ª definición (por comprensión):
comprueba :: [[Int]] -> Bool
comprueba xss = and [or [even x | x <- xs] | xs <- xss]

-- 2ª definición (por recursión):
compruebaR :: [[Int]] -> Bool
compruebaR [] = True
compruebaR (xs:xss) = tienePar xs && compruebaR xss

-- (tienePar xs) se verifica si xs contiene algún número par.
tienePar  :: [Int] -> Bool
tienePar []     = False
```

```haskell
tienePar (x:xs) = even x || tienePar xs

-- 3ª definición (por plegado):
compruebaP :: [[Int]] -> Bool
compruebaP = foldr ((&&) . tienePar) True

-- (tieneParP xs) se verifica si xs contiene algún número par.
tieneParP  :: [Int] -> Bool
tieneParP = foldr ((||) . even) False

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    pertenece :: Ord a => a -> [a] -> Bool
-- tal que (pertenece x ys) se verifica si x pertenece a la lista
-- ordenada creciente, finita o infinita, ys. Por ejemplo,
--    pertenece 22 [1,3,22,34]  ==  True
--    pertenece 22 [1,3,34]     ==  False
--    pertenece 23 [1,3..]      ==  True
--    pertenece 22 [1,3..]      ==  False
-- ---------------------------------------------------------------------

pertenece :: Ord a => a -> [a] -> Bool
pertenece _ [] = False
pertenece x (y:ys) | x >  y     = pertenece x ys
                   | x == y     = True
                   | otherwise = False

-- La definición de pertenece puede simplificarse
pertenece' :: Ord a => a -> [a] -> Bool
pertenece' x ys = x `elem` takeWhile (<= x) ys
```

1.3. Examen 3 (15 de marzo de 2010)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- 3º examen de evaluación continua (15 de marzo de 2010)
-- ---------------------------------------------------------------------

import Test.QuickCheck

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

```haskell
-- Ejercicio 1.1.1. Definir, por recursión, la función
--    pares :: [Int] -> [Int]
-- tal que (pares xs) es la lista de los elementos pares de xs. Por
-- ejemplo,
--    pares [2,5,7,4,6,8,9]    ==  [2,4,6,8]
-- ---------------------------------------------------------------------

pares   :: [Int] -> [Int]
pares [] = []
pares (x:xs) | even x    = x : pares xs
             | otherwise = pares xs

-- ---------------------------------------------------------------------
-- Ejercicio 1.1.2. Definir, por recursión, la función
--    impares :: [Int] -> [Int]
-- tal que (impares xs) es la lista de los elementos impares de xs. Por
-- ejemplo,
--    impares [2,5,7,4,6,8,9]  ==  [5,7,9]
-- ---------------------------------------------------------------------

impares   :: [Int] -> [Int]
impares [] = []
impares (x:xs) | odd x     = x : impares xs
               | otherwise = impares xs

-- ---------------------------------------------------------------------
-- Ejercicio 1.1.3. Definir, por recursión, la función
--    suma :: [Int] -> Int
-- tal que (suma xs) es la suma de los elementos de xs. Por ejemplo,
--    suma [2,5,7,4,6,8,9]  ==  41
-- ---------------------------------------------------------------------

suma :: [Int] -> Int
suma []     = 0
suma (x:xs) = x + suma xs

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Comprobar con QuickCheck que la suma de la suma de
-- (pares xs) y la suma de (impares xs) es igual que la suma de xs.
-- ---------------------------------------------------------------------
```

```haskell
-- La propiedad es
prop_pares :: [Int] -> Bool
prop_pares xs =
    suma (pares xs) + suma (impares xs) == suma xs

-- La comprobación es
--    ghci> quickCheck prop_pares
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 1.3 Demostrar por inducción que que la suma de la suma de
-- (pares xs) y la suma de (impares xs) es igual que la suma de xs.
-- ---------------------------------------------------------------------

{-
 Demostración:
 La propiedad que hay que demostrar es
    suma (pares xs) + suma (impares xs) = suma xs

 Caso base: Hay que demostrar que
    suma (pares []) + suma (impares []) = suma []
 En efecto,
    suma (pares []) + suma (impares [])
    = suma [] + suma []                     [por pares.1 e impares.1]
    = 0 + 0                                 [por suma.1]
    = 0                                     [por aritmética]
    = suma []                               [por suma.1]

 Paso de inducción: Se supone que la hipótesis de inducción
    suma (pares xs) + suma (impares xs) = suma xs
 Hay que demostrar que
    suma (pares (x:xs)) + suma (impares (x:xs)) = suma (x:xs)
 Lo demostraremos distinguiendo dos casos

 Caso 1: Supongamos que x es par. Entonces,
     suma (pares (x:xs)) + suma (impares (x:xs))
     = suma (x:pares xs) + suma (impares xs)       [por pares.2, impares.3]
     = x + suma (pares xs) + suma (impares xs)     [por suma.2]
     = x + suma xs                                 [por hip. de inducción]
```

```haskell
     = suma (x:xs)                                [por suma.2]

 Caso 1: Supongamos que x es impar. Entonces,
     suma (pares (x:xs)) + suma (impares (x:xs))
     = suma (pares xs) + suma (x:impares xs)      [por pares.3, impares.2]
     = suma (pares xs) + x + suma (impares xs)    [por suma.2]
     = x + suma xs                                [por hip. de inducción]
     = suma (x:xs)                                [por suma.2]
-}

-- ---------------------------------------------------------------------
-- Ejercicio 2.1.1. Definir, por recursión, la función
--    duplica :: [a] -> [a]
-- tal que (duplica xs) es la lista obtenida duplicando los elementos de
-- xs. Por ejemplo,
--    duplica  [7,2,5]  ==  [7,7,2,2,5,5]
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 2.1.2. Definir, por recursión, la función
--    longitud :: [a] -> Int
-- tal que (longitud xs) es el número de elementos de xs. Por ejemplo,
--    longitud [7,2,5]  ==  3
-- ---------------------------------------------------------------------

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

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Comprobar con QuickCheck que (longitud (duplica xs))
-- es el doble de (longitud xs), donde xs es una lista de números
-- enteros.
-- ---------------------------------------------------------------------

-- La propiedad es
```

```haskell
prop_duplica :: [Int] -> Bool
prop_duplica xs =
    longitud (duplica xs) == 2 * longitud xs

-- La comprobación es
--    ghci> quickCheck prop_duplica
--    OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 2.3. Demostrar por inducción que la longitud de
-- (duplica xs) es el doble de la longitud de xs.
-- ---------------------------------------------------------------------

{-
 Demostración: Hay que demostrar que
    longitud (duplica xs) = 2 * longitud xs
 Lo haremos por inducción en xs.

 Caso base: Hay que demostrar que
    longitud (duplica []) = 2 * longitud []
 En efecto
    longitud (duplica xs)
    = longitud []            [por duplica.1]
    = 0                      [por longitud.1]
    = 2 * 0                  [por aritmética]
    = longitud []            [por longitud.1]

 Paso de inducción: Se supone la hipótesis de inducción
    longitud (duplica xs) = 2 * longitud xs
 Hay que demostrar que
    longitud (duplica (x:xs)) = 2 * longitud (x:xs)
 En efecto,
    longitud (duplica (x:xs))
    = longitud (x:x:duplica xs)       [por duplica.2]
    = 1 + longitud (x:duplica xs)     [por longitud.2]
    = 1 + 1 + longitud (duplica xs)   [por longitud.2]
    = 1 + 1 + 2*(longitud xs)         [por hip. de inducción]
    = 2 * (1 + longitud xs)           [por aritmética]
    = 2 * longitud (x:xs)             [por longitud.2]
-}
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir la función
--    listasMayores :: [[Int]] -> [[Int]]
-- tal que (listasMayores xss) es la lista de las listas de xss de mayor
-- suma. Por ejemplo,
--    ghci> listasMayores [[1,3,5],[2,7],[1,1,2],[3],[5]]
--    [[1,3,5],[2,7]]
-- ---------------------------------------------------------------------

listasMayores :: [[Int]] -> [[Int]]
listasMayores xss = [xs | xs <- xss, sum xs == m]
    where m = maximum [sum xs | xs <- xss]

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Comprobar con QuickCheck que todas las listas de
-- (listasMayores xss) tienen la misma suma.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_listasMayores :: [[Int]] -> Bool
prop_listasMayores xss =
    iguales [sum xs | xs <- listasMayores xss]

-- (iguales xs) se verifica si todos los elementos de xs son
-- iguales. Por ejemplo,
--    iguales [2,2,2]  ==  True
--    iguales [2,3,2]  ==  False
iguales :: Eq a => [a] -> Bool
iguales (x1:x2:xs) = x1 == x2 && iguales (x2:xs)
iguales _          = True

-- La comprobación es
--    ghci> quickCheck prop_listasMayores
--    OK, passed 100 tests.
```

1.4. Examen 4 (12 de abril de 2010)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- 4º examen de evaluación continua (12 de abril de 2010)
```

```haskell
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Ejercicio 1.1. En los apartados de este ejercicio se usará el tipo de
-- árboles binarios definidos como sigue
--    data Arbol a = Hoja
--                 | Nodo a (Arbol a) (Arbol a)
--                 deriving (Show, Eq)
-- En los ejemplos se usará el siguiente árbol
--    ejArbol :: Arbol Int
--    ejArbol = Nodo 2
--                   (Nodo 5
--                         (Nodo 3 Hoja Hoja)
--                         (Nodo 7 Hoja Hoja))
--                   (Nodo 4 Hoja Hoja)
--
-- Definir por recursión la función
--    sumaArbol :: Num a => Arbol a -> a
-- tal (sumaArbol x) es la suma de los valores que hay en el árbol
-- x. Por ejemplo,
--    sumaArbol ejArbol  ==  21
-- ---------------------------------------------------------------------

data Arbol a = Hoja
             | Nodo a (Arbol a) (Arbol a)
             deriving (Show, Eq)

ejArbol :: Arbol Int
ejArbol = Nodo 2
               (Nodo 5
                     (Nodo 3 Hoja Hoja)
                     (Nodo 7 Hoja Hoja))
               (Nodo 4 Hoja Hoja)

sumaArbol :: Num a => Arbol a -> a
sumaArbol Hoja = 0
sumaArbol (Nodo x i d) = x + sumaArbol i + sumaArbol d

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir por recursión la función
```

```haskell
--    nodos :: Arbol a -> [a]
-- tal que (nodos x) es la lista de los nodos del árbol x. Por ejemplo.
--    nodos ejArbol  ==  [2,5,3,7,4]
-- ---------------------------------------------------------------------

nodos :: Arbol a -> [a]
nodos Hoja = []
nodos (Nodo x i d) = x : nodos i ++ nodos d

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Demostrar por inducción que para todo árbol a,
-- sumaArbol a = sum (nodos a).
-- Indicar la propiedad de sum que se usa en la demostración.
-- ---------------------------------------------------------------------

{-
 Caso base: Hay que demostrar que
    sumaArbol Hoja = sum (nodos Hoja)
 En efecto,
    sumaArbol Hoja
    = 0                    [por sumaArbol.1]
    = sum []               [por suma.1]
    = sum (nodos Hoja)     [por nodos.1]

 Caso inductivo: Se supone la hipótesis de inducción
    sumaArbol i = sum (nodos i)
    sumaArbol d = sum (nodos d)
 Hay que demostrar que
    sumaArbol (Nodo x i d) = sum (nodos (Nodo x i d))
 En efecto,
    sumaArbol (Nodo x i d)
    = x + sumaArbol i + sumaArbol d        [por sumaArbol.2]
    = x + sum (nodos i) + sum (nodos d)    [por hip. de inducción]
    = x + sum (nodos i ++ nodos d)         [por propiedad de sum]
    = sum(x:(nodos i)++(nodos d))          [por sum.2]
    = sum (Nodos x i d)                    [por nodos.2]
-}

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. Definir la constante
```

```haskell
--    pares :: Int
-- tal que pares es la lista de todos los pares de números enteros
-- positivos ordenada según la suma de sus componentes y el valor de la
-- primera componente. Por ejemplo,
--    ghci> take 11 pares
--    [(1,1),(1,2),(2,1),(1,3),(2,2),(3,1),(1,4),(2,3),(3,2),(4,1),(1,5)]
-- ---------------------------------------------------------------------

pares :: [(Integer,Integer)]
pares = [(x,z-x) | z <- [1..], x <- [1..z-1]]

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir la constante
--    paresDestacados :: [(Integer,Integer)]
-- tal que paresDestadados es la lista de pares de números enteros (x,y)
-- tales que 11 divide a x+13y y 13 divide a x+11y.
-- ---------------------------------------------------------------------

paresDestacados :: [(Integer,Integer)]
paresDestacados = [(x,y) | (x,y) <- pares,
                           x+13*y `rem` 11 == 0,
                           x+11*y `rem` 13 == 0]

-- ---------------------------------------------------------------------
-- Ejercicio 2.3. Definir la constante
--    parDestacadoConMenorSuma :: Integer
-- tal que parDestacadoConMenorSuma es el par destacado con menor suma y
-- calcular su valor y su posición en la lista pares.
-- ---------------------------------------------------------------------

-- La definición es
parDestacadoConMenorSuma :: (Integer,Integer)
parDestacadoConMenorSuma = head paresDestacados

-- El valor es
--   ghci> parDestacadoConMenorSuma
--   (23,5)

-- La posición es
--    ghci> 1 + length (takeWhile (/=parDestacadoConMenorSuma) pares)
```

```haskell
--    374

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir la función
--    limite :: (Num a, Enum a, Num b, Ord b) => (a -> b) -> b -> b
-- tal que (limite f a) es el valor de f en el primer término x tal que
-- para todo y entre x+1 y x+100, el valor absoluto de f(y)-f(x) es
-- menor que a. Por ejemplo,
--    limite (\n -> (2*n+1)/(n+5)) 0.001  ==  1.9900110987791344
--    limite (\n -> (1+1/n)**n) 0.001     ==  2.714072874546881
-- ---------------------------------------------------------------------

limite :: (Num a, Enum a, Num b, Ord b) => (a -> b) -> b -> b
limite f a =
    head [f x | x <- [1..],
                maximum [abs(f y - f x) | y <- [x+1..x+100]] < a]

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir la función
--    esLimite :: (Num a, Enum a, Num b, Ord b) =>
--                (a -> b) -> b -> b -> Bool
-- tal que (esLimite f b a) se verifica si existe un x tal que para todo
-- y entre x+1 y x+100, el valor absoluto de f(y)-b es menor que a. Por
-- ejemplo,
--    esLimite (\n -> (2*n+1)/(n+5)) 2 0.01     ==  True
--    esLimite (\n -> (1+1/n)**n) (exp 1) 0.01  ==  True
-- ---------------------------------------------------------------------

esLimite :: (Num a, Enum a, Num b, Ord b) => (a -> b) -> b -> b -> Bool
esLimite f b a =
    not (null [x | x <- [1..],
                   maximum [abs(f y - b) | y <- [x+1..x+100]] < a])
```

1.5. Examen 5 (17 de mayo de 2010)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- 5º examen de evaluación continua (17 de mayo de 2010)
-- ---------------------------------------------------------------------

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

```haskell
-- Ejercicio 1.1. Definir Haskell la función
--    primo :: Int -> Integer
-- tal que (primo n) es el n-ésimo número primo. Por ejemplo,
--    primo 5 = 11
-- ---------------------------------------------------------------------

primo :: Int -> Integer
primo n = primos !! (n-1)

-- primos es la lista de los números primos. Por ejemplo,
--    take 10 primos  ==  [2,3,5,7,11,13,17,19,23,29]
primos :: [Integer]
primos = 2 : [n | n <- [3,5..], esPrimo n]

-- (esPrimo n) se verifica si n es primo.
esPrimo :: Integer-> Bool
esPrimo n = [x | x <- [1..n], rem n x == 0] == [1,n]

-- ---------------------------------------------------------------------
-- Ejercicio 1.2. Definir la función
--    sumaCifras :: Integer -> Integer
-- tal que (sumaCifras n) es la suma de las cifras del número n. Por
-- ejemplo,
--    sumaCifras 325 = 10
-- ---------------------------------------------------------------------

sumaCifras :: Integer -> Integer
sumaCifras n
    | n < 10    = n
    | otherwise = sumaCifras(div n 10) + n `rem` 10

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Definir la función
--    primosSumaPar :: Int -> [Integer]
-- tal que (primosSumaPar n) es el conjunto de elementos del conjunto de
-- los n primeros primos tales que la suma de sus cifras es par. Por
-- ejemplo,
--    primosSumaPar 10 = [2,11,13,17,19]
-- ---------------------------------------------------------------------
```

```haskell
primosSumaPar :: Int -> [Integer]
primosSumaPar n =
   [x | x <- take n primos, even (sumaCifras x)]

-- ---------------------------------------------------------------------
-- Ejercicio 1.4. Definir la función
--    numeroPrimosSumaPar :: Int -> Int
-- tal que (numeroPrimosSumaPar n) es la cantidad de elementos del
-- conjunto de los n primeros primos tales que la suma de sus cifras es
-- par. Por ejemplo,
--    numeroPrimosSumaPar 10 = 5
-- ---------------------------------------------------------------------

numeroPrimosSumaPar :: Int -> Int
numeroPrimosSumaPar = length . primosSumaPar

-- ---------------------------------------------------------------------
-- Ejercicio 1.5. Definir la función
--    puntos :: Int -> [(Int,Int)]
-- tal que (puntos n) es la lista de los puntos de la forma (x,y) donde x
-- toma los valores 0,10,20,...,10*n e y es la cantidad de elementos del
-- conjunto de los x primeros primos tales que la suma de sus cifras es
-- par. Por ejemplo,
--    puntos 5 = [(0,0),(10,5),(20,10),(30,17),(40,21),(50,23)]
-- ---------------------------------------------------------------------

puntos :: Int -> [(Int,Int)]
puntos n = [(i,numeroPrimosSumaPar i) | i <- [0,10..10*n]]
```

1.6. Examen 6 (21 de junio de 2010)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- 6º examen de evaluación continua (21 de junio de 2010)
-- ---------------------------------------------------------------------

import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    calculaPi :: Int -> Double
```

```haskell
-- tal que (calculaPi n) es la aproximación del número pi calculada
-- mediante la expresión
--    4*(1 - 1/3 + 1/5 - 1/7 ... 1/(2*n+1))
-- Por ejemplo,
--    calculaPi 3    ==  2.8952380952380956
--    calculaPi 300  ==  3.1449149035588526
-- Indicación: La potencia es **, por ejemplo 2**3 es 8.0.
-- ---------------------------------------------------------------------

calculaPi :: Int -> Double
calculaPi n = 4 * sum [(-1)**x/(2*x+1) | x <- [0..fromIntegral n]]

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. En la Olimpiada de Matemática del 2010 se planteó el
-- siguiente problema:
--   Una sucesión pucelana es una sucesión creciente de 16 números
--   impares positivos consecutivos, cuya suma es un cubo perfecto.
--   ¿Cuántas sucesiones pucelanas tienen solamente números de tres
--   cifras?
--
-- Definir la función
--    pucelanas :: [[Int]]
-- tal que pucelanas es la lista de las sucesiones pucelanas. Por
-- ejemplo,
--    ghci> head pucelanas
--    [17,19,21,23,25,27,29,31,33,35,37,39,41,43,45,47]
-- ---------------------------------------------------------------------

pucelanas :: [[Int]]
pucelanas = [[x,x+2..x+30] | x <- [1..],
                             esCubo (sum [x,x+2..x+30])]

-- (esCubo n) se verifica si n es un cubo. Por ejemplo,
--    esCubo 27  ==  True
--    esCubo 28  ==  False
esCubo x = y^3== x
    where y = ceiling (fromIntegral x ** (1/3))

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir la función
```

```haskell
--     pucelanasConNcifras :: Int -> [[Int]]
-- tal que (pucelanasConNcifras n) es la lista de las sucesiones
-- pucelanas que tienen sólamente números de n cifras. Por ejemplo,
--     ghci> pucelanasConNcifras 2
--     [[17,19,21,23,25,27,29,31,33,35,37,39,41,43,45,47]]
-- ---------------------------------------------------------------------

pucelanasConNcifras :: Int -> [[Int]]
pucelanasConNcifras n = [[x,x+2..x+30] | x <- [10^(n-1)+1..10^n-31],
                                         esCubo (sum [x,x+2..x+30])]

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. Calcular cuántas sucesiones pucelanas tienen solamente
-- números de tres cifras.
-- ---------------------------------------------------------------------

-- El cálculo es
--     ghci> length (pucelanasConNcifras 3)
--     3

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--     inflexion :: Ord a => [a] -> Maybe a
-- tal que (inflexion xs) es el primer elemento de la lista en donde se
-- cambia de creciente a decreciente o de decreciente a creciente y
-- Nothing si no se cambia. Por ejemplo,
--     inflexion [2,2,3,5,4,6]    ==  Just 4
--     inflexion [9,8,6,7,10,10]  ==  Just 7
--     inflexion [2,2,3,5]        ==  Nothing
--     inflexion [5,3,2,2]        ==  Nothing
-- ---------------------------------------------------------------------

inflexion :: Ord a => [a] -> Maybe a
inflexion (x:y:zs)
    | x <  y = decreciente (y:zs)
    | x == y = inflexion (y:zs)
    | x >  y = creciente (y:zs)
inflexion _  = Nothing

-- (creciente xs) es el segundo elemento de la primera parte creciente
```

```
-- de xs y Nothing, en caso contrario. Por ejemplo,
--    creciente [4,3,5,6]  ==  Just 5
--    creciente [4,3,5,2,7]  ==  Just 5
--    creciente [4,3,2]  ==  Nothing
creciente (x:y:zs)
    | x <  y     = Just y
    | otherwise = creciente (y:zs)
creciente _      = Nothing

-- (decreciente xs) es el segundo elemento de la primera parte
-- decreciente de xs y Nothing, en caso contrario. Por ejemplo,
--    decreciente [4,2,3,1,0]  ==  Just 2
--    decreciente [4,5,3,1,0]  ==  Just 3
--    decreciente [4,5,7]      ==  Nothing
decreciente (x:y:zs)
    | x >  y     = Just y
    | otherwise = decreciente (y:zs)
decreciente _    = Nothing
```

1.7. Examen 7 (5 de julio de 2010)

```
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- Examen de la 1ª convocatoria (5 de julio de 2010)
-- ---------------------------------------------------------------------

import Test.QuickCheck
import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    numeroCerosFactorial :: Integer -> Integer
-- tal que (numeroCerosFactorial n) es el número de ceros con los que
-- termina el factorial de n. Por ejemplo,
--    numeroCerosFactorial 17 == 3
-- ---------------------------------------------------------------------

numeroCerosFactorial :: Integer -> Integer
numeroCerosFactorial n = numeroCeros (product [1..n])

-- (numeroCeros x) es el número de ceros con los que termina x. Por
```

```haskell
-- ejemplo,
--    numeroCeros 35400  ==  2
numeroCeros :: Integer -> Integer
numeroCeros x | mod x 10 /= 0 = 0
              | otherwise     = 1 + numeroCeros (div x 10)

-- ---------------------------------------------------------------------
-- Ejercicio 2. Las matrices pueden representarse mediante una lista de
-- listas donde cada una de las lista representa una fila  de la
-- matriz. Por ejemplo, la matriz
--    |1 0 -2|
--    |0 3 -1|
-- puede representarse por [[1,0,-2],[0,3,-1]]. Definir la función
--    producto :: Num t => [[t]] -> [[t]] -> [[t]]
-- tal que (producto a b) es el producto de las matrices a y b. Por
-- ejemplo,
--    ghci> producto [[1,0,-2],[0,3,-1]] [[0,3],[-2,-1],[0,4]]
--    [[0,-5],[-6,-7]]
-- ---------------------------------------------------------------------

producto :: Num t => [[t]] -> [[t]] -> [[t]]
producto a b =
    [[sum [x*y | (x,y) <- zip fil col] | col <- transpose b] | fil <- a]

-- ---------------------------------------------------------------------
-- Ejercicio 3. El ejercicio 4 de la Olimpiada Matemáticas de 1993 es el
-- siguiente:
--    Demostrar que para todo número primo p distinto de 2 y de 5,
--    existen infinitos múltiplos de p de la forma 1111......1 (escrito
--    sólo con unos).
-- Definir la función
--    multiplosEspeciales :: Integer -> Int -> [Integer]
-- tal que (multiplosEspeciales p n) es una lista de n múltiplos p de la
-- forma 1111...1 (escrito sólo con unos), donde p es un número primo
-- distinto de 2 y 5. Por ejemplo,
--    multiplosEspeciales 7 2  ==  [111111,111111111111]
-- ---------------------------------------------------------------------

-- 1ª definición:
multiplosEspeciales :: Integer -> Int -> [Integer]
```

```
multiplosEspeciales p n = take n [x | x <- unos, mod x p == 0]

-- unos es la lista de los números de la forma 111...1 (escrito sólo con
-- unos). Por ejemplo,
--    take 5 unos  ==  [1,11,111,1111,11111]
unos :: [Integer]
unos = 1 : [10*x+1 | x <- unos]

-- Otra definición no recursiva de unos es
unos' :: [Integer]
unos' = [div (10^n-1) 9 | n <- [1..]]

-- 2ª definición:
multiplosEspeciales2 :: Integer -> Int -> [Integer]
multiplosEspeciales2 p n =
    [div (10^((p-1)*x)-1) 9 | x <- [1..fromIntegral n]]

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    recorridos :: [a] -> [[a]]
-- tal que (recorridos xs) es la lista de todos los posibles recorridos
-- por el grafo cuyo conjunto de vértices es xs y cada vértice se
-- encuentra conectado con todos los otros y los recorridos pasan por
-- todos los vértices una vez y terminan en el vértice inicial. Por
-- ejemplo,
--    ghci> recorridos [2,5,3]
--    [[2,5,3,2],[5,2,3,5],[3,5,2,3],[5,3,2,5],[3,2,5,3],[2,3,5,2]]
-- Indicación: No importa el orden de los recorridos en la lista.
-- ---------------------------------------------------------------------

recorridos :: [a] -> [[a]]
recorridos xs = [(y:ys)++[y] | (y:ys) <- permutations xs]
```

1.8. Examen 8 (15 de septiembre de 2010)

```
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- Examen de la 2ª convocatoria (15 de septiembre de 2010)
-- ---------------------------------------------------------------------

import Test.QuickCheck
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 1.1. Definir la función
--    diagonal :: [[a]] -> [a]
-- tal que (diagonal m) es la diagonal de la matriz m. Por ejemplo,
--    diagonal [[3,5,2],[4,7,1],[6,9,0]]  ==  [3,7,0]
--    diagonal [[3,5,2],[4,7,1]]          ==  [3,7]
-- ---------------------------------------------------------------------

-- 1ª definición (por recursión):
diagonal :: [[a]] -> [a]
diagonal ((x1:_):xs) = x1 : diagonal [tail x | x <- xs]
diagonal _ = []

-- Segunda definición (sin recursión):
diagonal2 :: [[a]] -> [a]
diagonal2 = flip (zipWith (!!)) [0..]

-- ---------------------------------------------------------------------.
-- Ejercicio 1.2. Definir la función
--    matrizDiagonal ::  Num a => [a] -> [[a]]
-- tal que (matrizDiagonal xs) es la matriz cuadrada cuya diagonal es el
-- vector xs y los restantes elementos son iguales a cero. Por ejemplo,
--    matrizDiagonal [2,5,3]  ==  [[2,0,0],[0,5,0],[0,0,3]]
-- ---------------------------------------------------------------------

matrizDiagonal :: Num a => [a] -> [[a]]
matrizDiagonal [] = []
matrizDiagonal (x:xs) =
    (x: [0 | _ <- xs]) : [0:zs | zs <- ys]
    where ys = matrizDiagonal xs

-- ---------------------------------------------------------------------
-- Ejercicio 1.3. Comprobar con QuickCheck si se verifican las
-- siguientes propiedades:
-- 1. Para cualquier lista xs, (diagonal (matrizDiagonal xs)) es igual a
--    xs.
-- 2. Para cualquier matriz m, (matrizDiagonal (diagonal m)) es igual a
--    m.
-- ---------------------------------------------------------------------
```

```haskell
-- La primera propiedad es
prop_diagonal1 :: [Int] -> Bool
prop_diagonal1 xs =
    diagonal (matrizDiagonal xs) == xs

-- La comprobación es
--    ghci> quickCheck prop_diagonal1
--    +++ OK, passed 100 tests.

-- La segunda propiedad es
prop_diagonal2 :: [[Int]] -> Bool
prop_diagonal2 m =
    matrizDiagonal (diagonal m) == m

-- La comprobación es
--    ghci> quickCheck prop_diagonal2
--    *** Failed! Falsifiable (after 4 tests and 5 shrinks):
--    [[0,0]]
-- lo que indica que la propiedad no se cumple y que [[0,0]] es un
-- contraejemplo,

-- ---------------------------------------------------------------------
-- Ejercicio 2.1. El enunciado del problema 1 de la Fase nacional de la
-- Olimpiada Matemática Española del 2009 dice:
--    Hallar todas las sucesiones finitas de n números naturales
--    consecutivos a1, a2, ..., an, con n ≥ 3, tales que
--    a1 + a2 + ... + an = 2009.
--
-- En este ejercicio vamos a resolver el problema con Haskell.
--
-- Definir la función
--    sucesionesConSuma :: Int -> [[Int]]
-- tal que (sucesionesConSuma x) es la lista de las sucesiones finitas
-- de n números naturales consecutivos a1, a2, ..., an, con n ≥ 3, tales
-- que
--    a1 + a2 + ... + an = x.
-- Por ejemplo.
--    sucesionesConSuma 9   ==  [[2,3,4]]
--    sucesionesConSuma 15  ==  [[1,2,3,4,5],[4,5,6]]
```

```haskell
-- ---------------------------------------------------------------------

-- 1ª definición:
sucesionesConSuma :: Int -> [[Int]]
sucesionesConSuma x =
    [[a..b] | a <- [1..x], b <- [a+2..x], sum [a..b] == x]

-- 2ª definición (con la fórmula de la suma de las progresiones
-- aritméticas):
sucesionesConSuma' :: Int -> [[Int]]
sucesionesConSuma' x =
    [[a..b] | a <- [1..x], b <- [a+2..x], (a+b)*(b-a+1) `div` 2 == x]

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Resolver el problema de la Olimpiada con la función
-- sucesionesConSuma.
-- ---------------------------------------------------------------------

-- Las soluciones se calculan con
--    ghci> sucesionesConSuma' 2009
--    [[17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,
--      38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,
--      59,60,61,62,63,64,65],
--     [29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,
--      50,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69],
--     [137,138,139,140,141,142,143,144,145,146,147,148,149,150],
--     [284,285,286,287,288,289,290]]
-- Por tanto, hay 4 soluciones.

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir la función
--    sumasNcuadrados :: Int -> Int -> [[Int]]
-- tal que (sumasNcuadrados x n) es la lista de las descomposiciones de
-- x en sumas decrecientes de n cuadrados. Por ejemplo,
--    sumasNcuadrados 10 4  ==  [[3,1,0,0],[2,2,1,1]]
-- ---------------------------------------------------------------------

sumasNcuadrados :: Int -> Int -> [[Int]]
sumasNcuadrados x 1 | a^2 == x  = [[a]]
                    | otherwise = []
```

```haskell
    where a = ceiling (sqrt (fromIntegral x))
sumasNcuadrados x n =
    [a:y:ys | a <- [x',x'-1..0],
              (y:ys) <- sumasNcuadrados (x-a^2) (n-1),
              y <= a]
    where x' = ceiling (sqrt (fromIntegral x))

-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Definir la función
--    numeroDeCuadrados :: Int -> Int
-- tal que (numeroDeCuadrados x) es el menor número de cuadrados que se
-- necesita para escribir x como una suma de cuadrados. Por ejemplo,
--    numeroDeCuadrados 6   ==  3
--    sumasNcuadrados 6 3   ==  [[2,1,1]]
-- ---------------------------------------------------------------------

numeroDeCuadrados :: Int -> Int
numeroDeCuadrados x = head [n | n <- [1..], sumasNcuadrados x n /= []]

-- ---------------------------------------------------------------------
-- Ejercicio 3.3. Calcular el menor número n tal que todos los números
-- de 0 a 100 pueden expresarse como suma de n cuadrados.
-- ---------------------------------------------------------------------

-- El cálculo de n es
--    ghci> maximum [numeroDeCuadrados x | x <- [0..100]]
--    4

-- ---------------------------------------------------------------------
-- Ejercicio 3.4. Comprobar con QuickCheck si todos los números
-- positivos pueden expresarse como suma de n cuadrados (donde n es el
-- número calculado anteriormente).
-- ---------------------------------------------------------------------

-- La propiedad es
prop_numeroDeCuadrados x =
    x >= 0 ==> numeroDeCuadrados x <= 4

-- La comprobación es
--    ghci> quickCheck prop_numeroDeCuadrados
```

```
--    OK, passed 100 tests.
```

1.9. Examen 9 (17 de diciembre de 2010)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- Examen de la 3ª convocatoria (17 de diciembre de 2010)
-- ---------------------------------------------------------------------

import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    ullman :: (Num a, Ord a) => a -> Int -> [a] -> Bool
-- tal que (ullman t k xs) se verifica si xs tiene un subconjunto con k
-- elementos cuya suma sea menor que t. Por ejemplo,
--    ullman 9 3 [1..10] == True
--    ullman 5 3 [1..10] == False
-- ---------------------------------------------------------------------

-- 1ª solución (corta y eficiente)
ullman :: (Ord a, Num a) => a -> Int -> [a] -> Bool
ullman t k xs = sum (take k (sort xs)) < t

-- 2ª solución (larga e ineficiente)
ullman2 :: (Num a, Ord a) => a -> Int -> [a] -> Bool
ullman2 t k xs =
    [ys | ys <- subconjuntos xs, length ys == k, sum ys < t] /= []

-- (subconjuntos xs) es la lista de los subconjuntos de xs. Por
-- ejemplo,
--    subconjuntos "bc"  ==  ["","c","b","bc"]
--    subconjuntos "abc" ==  ["","c","b","bc","a","ac","ab","abc"]
subconjuntos :: [a] -> [[a]]
subconjuntos [] = [[]]
subconjuntos (x:xs) = zss++[x:ys | ys <- zss]
    where zss = subconjuntos xs

-- Los siguientes ejemplos muestran la diferencia en la eficencia:
--    ghci> ullman 9 3 [1..20]
--    True
```

```haskell
--    (0.02 secs, 528380 bytes)
--    ghci> ullman2 9 3 [1..20]
--    True
--    (4.08 secs, 135267904 bytes)
--    ghci> ullman 9 3 [1..100]
--    True
--    (0.02 secs, 526360 bytes)
--    ghci> ullman2 9 3 [1..100]
--      C-c C-cInterrupted.
--    Agotado

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    sumasDe2Cuadrados :: Integer -> [(Integer, Integer)]
-- tal que (sumasDe2Cuadrados n) es la lista de los pares de números
-- tales que la suma de sus cuadrados es n y el primer elemento del par
-- es mayor o igual que el segundo. Por ejemplo,
--    sumasDe2Cuadrados 25  ==  [(5,0),(4,3)]
-- ---------------------------------------------------------------------

-- 1ª definición:
sumasDe2Cuadrados_1 :: Integer -> [(Integer, Integer)]
sumasDe2Cuadrados_1 n =
    [(x,y) | x <- [n,n-1..0],
             y <- [0..x],
             x*x+y*y == n]

-- 2ª definición:
sumasDe2Cuadrados2 :: Integer -> [(Integer, Integer)]
sumasDe2Cuadrado_2 n =
    [(x,y) | x <- [a,a-1..0],
             y <- [0..x],
             x*x+y*y == n]
    where a = ceiling (sqrt (fromIntegral n))

-- 3ª definición:
sumasDe2Cuadrados3 :: Integer -> [(Integer, Integer)]
sumasDe2Cuadrado_3 n = aux (ceiling (sqrt (fromIntegral n))) 0
    where aux x y | x < y          = []
                  | x*x + y*y <  n = aux x (y+1)
```

```
                  | x*x + y*y == n = (x,y) : aux (x-1) (y+1)
                  | otherwise      = aux (x-1) y

-- Comparación
--    +----------+--------------+--------------+--------------+
--    | n        | 1ª definición | 2ª definición | 3ª definición |
--    +----------+--------------+--------------+--------------+
--    |      999 | 2.17 segs     |   0.02 segs   | 0.01 segs     |
--    | 48612265 |               | 140.38 segs   | 0.13 segs     |
--    +----------+--------------+--------------+--------------+

-- ---------------------------------------------------------------------
-- Ejercicio 3. Los árboles binarios pueden representarse mediante el
-- tipo de datos Arbol definido por
--    data Arbol a = Nodo (Arbol a) (Arbol a)
--                 | Hoja a
--                 deriving Show
-- Por ejemplo, los árboles
--    árbol1          árbol2       árbol3     árbol4
--       o              o            o                 o
--      / \            / \          / \               / \
--     1   o          o   3        o   3             o   1
--        / \        / \          / \               / \
--       2   3      1   2        1   4             2   3
-- se representan por
--    arbol1, arbol2, arbol3, arbol4 :: Arbol Int
--    arbol1 = Nodo (Hoja 1) (Nodo (Hoja 2) (Hoja 3))
--    arbol2 = Nodo (Nodo (Hoja 1) (Hoja 2)) (Hoja 3)
--    arbol3 = Nodo (Nodo (Hoja 1) (Hoja 4)) (Hoja 3)
--    arbol4 = Nodo (Nodo (Hoja 2) (Hoja 3)) (Hoja 1)
-- Definir la función
--    igualBorde :: Eq a => Arbol a -> Arbol a -> Bool
-- tal que (igualBorde t1 t2) se verifica si los bordes de los árboles
-- t1 y t2 son iguales. Por ejemplo,
--    igualBorde arbol1 arbol2  ==  True
--    igualBorde arbol1 arbol3  ==  False
--    igualBorde arbol1 arbol4  ==  False
-- ---------------------------------------------------------------------

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

```haskell
             | Hoja a
             deriving Show

arbol1, arbol2, arbol3, arbol4 :: Arbol Int
arbol1 = Nodo (Hoja 1) (Nodo (Hoja 2) (Hoja 3))
arbol2 = Nodo (Nodo (Hoja 1) (Hoja 2)) (Hoja 3)
arbol3 = Nodo (Nodo (Hoja 1) (Hoja 4)) (Hoja 3)
arbol4 = Nodo (Nodo (Hoja 2) (Hoja 3)) (Hoja 1)

igualBorde :: Eq a => Arbol a -> Arbol a -> Bool
igualBorde t1 t2 = borde t1 == borde t2

-- (borde t) es el borde del árbol t; es decir, la lista de las hojas
-- del árbol t leídas de izquierda a derecha. Por ejemplo,
--    borde arbol4  ==  [2,3,1]
borde :: Arbol a -> [a]
borde (Nodo i d) = borde i ++ borde d
borde (Hoja x)   = [x]

-- ---------------------------------------------------------------------
-- Ejercicio 4. (Basado en el problema 145 del Proyecto Euler).
-- Se dice que un número n es reversible si su última cifra es
-- distinta de 0 y la suma de n y el número obtenido escribiendo las
-- cifras de n en orden inverso es un número que tiene todas sus cifras
-- impares. Por ejemplo,
-- * 36  es reversible porque 36+63=99 tiene todas sus cifras impares,
-- * 409 es reversible porque 409+904=1313 tiene todas sus cifras
--       impares,
-- * 243 no es reversible porque 243+342=585 no tiene todas sus cifras
--       impares.
--
-- Definir la función
--    reversiblesMenores :: Int -> Int
-- tal que (reversiblesMenores n) es la cantidad de números reversibles
-- menores que n. Por ejemplo,
--    reversiblesMenores 10   == 0
--    reversiblesMenores 100  == 20
--    reversiblesMenores 1000 == 120
-- ---------------------------------------------------------------------
```

```haskell
-- (reversiblesMenores n) es la cantidad de números reversibles menores
-- que n. Por ejemplo,
--    reversiblesMenores 10   == 0
--    reversiblesMenores 100  == 20
--    reversiblesMenores 1000 == 120
reversiblesMenores :: Int -> Int
reversiblesMenores n = length [x | x <- [1..n-1], esReversible x]

-- (esReversible n) se verifica si n es reversible; es decir, si su
-- última cifra es distinta de 0 y la suma de n y el número obtenido
-- escribiendo las cifras de n en orden inverso es un número que tiene
-- todas sus cifras impares. Por ejemplo,
--    esReversible 36  == True
--    esReversible 409 == True
esReversible :: Int -> Bool
esReversible n = rem n 10 /= 0 && impares (cifras (n + (inverso n)))

-- (impares xs) se verifica si xs es una lista de números impares. Por
-- ejemplo,
--    impares [3,5,1] == True
--    impares [3,4,1] == False
impares :: [Int] -> Bool
impares xs = and [odd x | x <- xs]

-- (inverso n) es el número obtenido escribiendo las cifras de n en
-- orden inverso. Por ejemplo,
--    inverso 3034 == 4303
inverso :: Int -> Int
inverso n = read (reverse (show n))

-- (cifras n) es la lista de las cifras del número n. Por ejemplo,
--    cifras 3034 == [3,0,3,4]
cifras :: Int -> [Int]
cifras n = [read [x] | x <- show n]
```

2

Exámenes del grupo 2

María J. Hidalgo

2.1. Examen 1 (4 de diciembre de 2009)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- 1º examen de evaluación continua (4 de diciembre de 2009)
-- ---------------------------------------------------------------------

import Test.QuickCheck
import Data.Char
import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir, por composición, la función
--    insertaEnposicion :: a -> Int -> [a] -> [a]
-- tal que (insertaEnposicion x n xs) es la lista obtenida insertando x
-- en xs en la posición n. Por ejemplo,
--    insertaEnposicion 80 4 [1,2,3,4,5,6,7,8] == [1,2,3,80,4,5,6,7,8]
--    insertaEnposicion 'a' 1 "hola"            == "ahola"
-- ---------------------------------------------------------------------

insertaEnposicion :: a -> Int -> [a] -> [a]
insertaEnposicion x n xs = take (n-1) xs ++ [x] ++ drop (n-1) xs

-- ---------------------------------------------------------------------
-- Ejercicio 2. El algoritmo de Euclides para calcular el máximo común
-- divisor de dos números naturales a y b es el siguiente:
--    Si b = 0, entonces mcd(a,b) = a
```

```haskell
--    Si b > 0, entonces mcd(a,b) = mcd(b,c), donde c es el resto de
--              dividir a entre b
--
-- Definir la función
--    mcd :: Int -> Int -> Int
-- tal que (mcd a b) es el máximo común divisor de a y b calculado
-- usando el algoritmo de Euclides. Por ejemplo,
--    mcd 2 3      ==  1
--    mcd 12 30    ==  6
--    mcd 700 300  ==  100
-- ---------------------------------------------------------------------

mcd :: Int -> Int -> Int
mcd a 0 = a
mcd a b = mcd b (a `mod` b)

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir la función
--    esSubconjunto :: Eq a => [a] -> [a] -> Bool
-- tal que (esSubconjunto xs ys) se verifica si todos los elementos de
-- xs son también elementos de ys. Por ejemplo,
--    esSubconjunto [3,5,2,1,1,1,6,3] [1,2,3,5,6,7]   == True
--    esSubconjunto [3,2,1,1,1,6,3] [1,2,3,5,6,7]     == True
--    esSubconjunto [3,2,1,8,1,6,3] [1,2,3,5,6,7]     == False
-- ---------------------------------------------------------------------

-- 1ª definición (por recursión):
esSubconjunto :: Eq a => [a] -> [a] -> Bool
esSubconjunto [] ys     = True
esSubconjunto (x:xs) ys = x `elem` ys && esSubconjunto xs ys

-- 2ª definición (por comprensión):
esSubconjunto2 :: Eq a => [a] -> [a] -> Bool
esSubconjunto2 xs ys = and [x `elem` ys | x <- xs]

-- 3ª definición (por plegado):
esSubconjunto3 :: Eq a => [a] -> [a] -> Bool
esSubconjunto3 xs ys = foldr (\ x -> (&&) (x `elem` ys)) True xs

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

```haskell
-- Ejercicio 3.2. Definir la función
--    igualConjunto :: Eq a => [a] -> [a] -> Bool
-- tal que (igualConjunto xs ys) se verifica si xs e ys son iguales como
-- conjuntos. Por ejemplo,
--    igualConjunto [3,2,1,8,1,6,3] [1,2,3,5,6,7]     == False
--    igualConjunto [3,2,1,1,1,6,3] [1,2,3,5,6,7]     == False
--    igualConjunto [3,2,1,1,1,6,3] [1,2,3,5,6]       == False
--    igualConjunto [3,2,1,1,1,6,3,5] [1,2,3,5,6]     == True
-- ---------------------------------------------------------------------

igualConjunto :: Eq a => [a] -> [a] -> Bool
igualConjunto xs ys = esSubconjunto xs ys && esSubconjunto ys xs

-- ---------------------------------------------------------------------
-- Ejercicio 4.1. Definir por comprensión la función
--    repiteC :: Int -> [a] -> [a]
-- tal que (repiteC n xs) es la lista que resulta de repetir cada
-- elemento de xs n veces. Por ejemplo,
--    repiteC 5 "Hola" == "HHHHHooooolllllaaaaa"
-- ---------------------------------------------------------------------

repiteC :: Int -> [a] -> [a]
repiteC n xs = [x | x <- xs, i <- [1..n]]

-- ---------------------------------------------------------------------
-- Ejercicio 4.2. Definir por recursión la función
--    repiteR :: Int -> [a] -> [a]
-- tal que (repiteR n xs) es la lista que resulta de repetir cada
-- elemento de xs n veces. Por ejemplo,
--    repiteR 5 "Hola" == "HHHHHooooolllllaaaaa"
-- ---------------------------------------------------------------------

repiteR :: Int -> [a] -> [a]
repiteR _ []     = []
repiteR n (x:xs) = replicate n x ++ repiteR n xs
```

2.2. Examen 2 (16 de marzo de 2010)

```haskell
-- Informática (1º del Grado en Matemáticas, Grupo 1)
-- 2º examen de evaluación continua (16 de marzo de 2010)
```

```haskell
-- ---------------------------------------------------------------------

import Test.QuickCheck
import Data.Char
import Data.List

-- ---------------------------------------------------------------------
-- Ejercicio 1. Probar por inducción que para toda lista xs:
--    length (reverse xs) = length xs
--
-- Nota: Las definiciones recursivas de length y reverse son:
--
--    length [] = 0                          -- length.1
--    length (x:xs) = 1 + length xs          -- length.2
--    reverse [] = []                        -- reverse.1
--    reverse (x:xs) = reverse xs ++ [x]     -- reverse.2
-- ---------------------------------------------------------------------

{-
La demostración es por inducción en xs.

Base: Supongamos que xs = []. Entonces,
   length (reverse xs)
   = length (reverse [])
   = length []                   [por reverse.1]
   = length xs.

Paso de inducción: Supongamos la hipótesis de inducción
   length (reverse xs) = length xs                 (H.I.)
y sea x un elemento cualquiera. Hay que demostrar que
   length (reverse (x:xs)) = length (x:xs)
En efecto,
   length (reverse (x:xs))
   = length (reverse xs ++ [x])         [por reverse.2]
   = length (reverse xs) + length [x]
   = length xs + 1                      [por H.I.]
   = length (x:xs)                      [por length.2]
-}

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

```haskell
-- Ejercicio 2.1. Definir, por recursión, la función
--    sumaVectores :: [Int] -> [Int] -> [Int]
-- tal que (sumaVectores v w) es la lista obtenida sumando los elementos
-- de v y w que ocupan las mismas posiciones. Por ejemplo,
--    sumaVectores [1,2,5,-6] [0,3,-2,9]  ==  [1,5,3,3]
-- ---------------------------------------------------------------------

-- 1ª definición (por comprensión)
sumaVectores :: [Int] -> [Int] -> [Int]
sumaVectores xs ys = [x+y | (x,y) <- zip xs ys]

-- 2ª definición (por recursión):
sumaVectores2 :: [Int] -> [Int] -> [Int]
sumaVectores2 [] _          = []
sumaVectores2 _ []          = []
sumaVectores2 (x:xs) (y:ys) = x+y : sumaVectores2 xs ys

-- ---------------------------------------------------------------------
-- Ejercicio 2.2. Definir, por recursión, la función
--    multPorEscalar :: Int -> [Int] -> [Int]
-- tal que (multPorEscalar x v) es la lista que resulta de multiplicar
-- todos los elementos de v por x. Por ejemplo,
--    multPorEscalar 4 [1,2,5,-6]  ==  [4,8,20,-24]
-- ---------------------------------------------------------------------

multPorEscalar :: Int -> [Int] -> [Int]
multPorEscalar _ [] = []
multPorEscalar n (x:xs) = n*x : multPorEscalar n xs

-- ---------------------------------------------------------------------
-- Ejercicio 2.3. Comprobar con QuickCheck que las operaciones
-- anteriores verifican la propiedad distributiva de multPorEscalar con
-- respecto a sumaVectores.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_distributiva :: Int -> [Int] -> [Int] -> Bool
prop_distributiva n xs ys =
    multPorEscalar n (sumaVectores xs ys) ==
    sumaVectores (multPorEscalar n xs) (multPorEscalar n ys)
```

```
-- La comprobación es
--    ghci> quickCheck prop_distributiva
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 2.4. Probar, por inducción, la propiedad anterior.
-- ---------------------------------------------------------------------

{-
La demostración es por inducción en xs.

Base: Supongamos que xs = []. Entonces,
   multPorEscalar n (sumaVectores xs ys)
   = multPorEscalar n (sumaVectores [] ys)
   = multPorEscalar n []
   = []
   = sumaVectores [] (multPorEscalar n ys)
   = sumaVectores (multPorEscalar n []) (multPorEscalar n ys)
   = sumaVectores (multPorEscalar n xs) (multPorEscalar n ys)

Paso de inducción: Supongamos la hipótesis de inducción
    multPorEscalar n (sumaVectores xs ys)
    = sumaVectores (multPorEscalar n xs) (multPorEscalar n ys) (H.I. 1)
Hay que demostrar que
    multPorEscalar n (sumaVectores (x:xs) ys)
    = sumaVectores (multPorEscalar n (x:xs)) (multPorEscalar n ys)
Lo haremos por casos en ys.

Caso 1: Supongamos que ys = []. Entonces,
   multPorEscalar n (sumaVectores xs ys)
   = multPorEscalar n (sumaVectores xs [])
   = multPorEscalar n []
   = []
   = sumaVectores (multPorEscalar n xs) []
   = sumaVectores (multPorEscalar n xs) (multPorEscalar n [])
   = sumaVectores (multPorEscalar n xs) (multPorEscalar n ys)

Caso 2: Para (y:ys). Entonces,
    multPorEscalar n (sumaVectores (x:xs) (y:ys))
```

```haskell
    = multPorEscalar n (x+y : sumaVectores xs ys)
        [por multPorEscalar.2]
    = n*(x+y) : multPorEscalar n (sumaVectores xs ys)
        [por multPorEscalar.2]
    = n*x+n*y : sumaVectores (multPorEscalar n xs) (multPorEscalar n ys)
        [por H.I. 1]
    = sumaVectores (n*x : multPorEscalar n xs) (n*y : multPorEscalar n ys)
        [por sumaVectores.2]
    = sumaVectores (multPorEscalar n (x:xs)) (multPorEscalar n (y:ys))
        [por multPorEscalar.2]
-}

-- ---------------------------------------------------------------------
-- Ejercicio 3. Consideremos los árboles binarios definidos como sigue
--    data Arbol a = H
--                 | N a (Arbol a) (Arbol a)
--                 deriving (Show, Eq)
--
-- Definir la función
--    mapArbol :: (a -> b) -> Arbol a -> Arbol b
-- tal que (mapArbol f x) es el árbol que resulta de sustituir cada nodo
-- n del árbol x por (f n). Por ejemplo,
--    ghci> mapArbol (+1) (N 9 (N 3 (N 2 H H) (N 4 H H)) (N 7 H H))
--    N 10 (N 8 H H) (N 4 (N 5 H H) (N 3 H H))
-- ---------------------------------------------------------------------

data Arbol a = H
             | N a (Arbol a) (Arbol a)
             deriving (Show, Eq)

mapArbol :: (a -> b) -> Arbol a -> Arbol b
mapArbol _ H         = H
mapArbol f (N x i d) = N (f x) (mapArbol f i) (mapArbol f d)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir, por comprensión, la función
--    mayorExpMenor :: Int -> Int -> Int
-- tal que (mayorExpMenor a b) es el menor n tal que a^n es mayor que
-- b. Por ejemplo,
--    mayorExpMenor 2 1000  ==  10
```

```
--    mayorExpMenor 9 7     ==   1
-- ---------------------------------------------------------------------

mayorExpMenor :: Int -> Int -> Int
mayorExpMenor a b =
    head [n | n <- [0..], a^n > b]
```

2.3. Examen 3 (5 de julio de 2010)

El examen es común con el del grupo 1 (ver página 25).

2.4. Examen 4 (15 de septiembre de 2010)

El examen es común con el del grupo 1 (ver página 27).

2.5. Examen 5 (17 de diciembre de 2010)

El examen es común con el del grupo 1 (ver página 32).

Apéndice A

Resumen de funciones predefinidas de Haskell

1. `x + y` es la suma de `x` e `y`.
2. `x - y` es la resta de `x` e `y`.
3. `x / y` es el cociente de `x` entre `y`.
4. `x ^y` es `x` elevado a `y`.
5. `x == y` se verifica si `x` es igual a `y`.
6. `x /= y` se verifica si `x` es distinto de `y`.
7. `x < y` se verifica si `x` es menor que `y`.
8. `x <= y` se verifica si `x` es menor o igual que `y`.
9. `x > y` se verifica si `x` es mayor que `y`.
10. `x >= y` se verifica si `x` es mayor o igual que `y`.
11. `x && y` es la conjunción de `x` e `y`.
12. `x || y` es la disyunción de `x` e `y`.
13. `x:ys` es la lista obtenida añadiendo `x` al principio de `ys`.
14. `xs ++ ys` es la concatenación de `xs` e `ys`.
15. `xs !! n` es el elemento `n`–ésimo de `xs`.
16. `f . g` es la composición de `f` y `g`.
17. `abs x` es el valor absoluto de `x`.
18. `and xs` es la conjunción de la lista de booleanos `xs`.
19. `ceiling x` es el menor entero no menor que `x`.
20. `chr n` es el carácter cuyo código ASCII es `n`.
21. `concat xss` es la concatenación de la lista de listas `xss`.
22. `const x y` es `x`.

23. `curry f` es la versión curryficada de la función `f`.
24. `div x y` es la división entera de `x` entre `y`.
25. `drop n xs` borra los `n` primeros elementos de `xs`.
26. `dropWhile p xs` borra el mayor prefijo de `xs` cuyos elementos satisfacen el predicado `p`.
27. `elem x ys` se verifica si `x` pertenece a `ys`.
28. `even x` se verifica si `x` es par.
29. `filter p xs` es la lista de elementos de la lista `xs` que verifican el predicado `p`.
30. `flip f x y` es `f y x`.
31. `floor x` es el mayor entero no mayor que `x`.
32. `foldl f e xs` pliega `xs` de izquierda a derecha usando el operador f y el valor inicial `e`.
33. `foldr f e xs` pliega `xs` de derecha a izquierda usando el operador f y el valor inicial `e`.
34. `fromIntegral x` transforma el número entero `x` al tipo numérico correspondiente.
35. `fst p` es el primer elemento del par `p`.
36. `gcd x y` es el máximo común divisor de de `x` e `y`.
37. `head xs` es el primer elemento de la lista `xs`.
38. `init xs` es la lista obtenida eliminando el último elemento de `xs`.
39. `iterate f x` es la lista `[x, f(x), f(f(x)), ...]`.
40. `last xs` es el último elemento de la lista `xs`.
41. `length xs` es el número de elementos de la lista `xs`.
42. `map f xs` es la lista obtenida aplicado `f` a cada elemento de `xs`.
43. `max x y` es el máximo de `x` e `y`.
44. `maximum xs` es el máximo elemento de la lista `xs`.
45. `min x y` es el mínimo de `x` e `y`.
46. `minimum xs` es el mínimo elemento de la lista `xs`.
47. `mod x y` es el resto de `x` entre `y`.
48. `not x` es la negación lógica del booleano `x`.
49. `noElem x ys` se verifica si `x` no pertenece a `ys`.
50. `null xs` se verifica si `xs` es la lista vacía.
51. `odd x` se verifica si `x` es impar.
52. `or xs` es la disyunción de la lista de booleanos `xs`.
53. `ord c` es el código ASCII del carácter `c`.

54. `product xs` es el producto de la lista de números `xs`.
55. `read c` es la expresión representada por la cadena `c`.
56. `rem x y` es el resto de `x` entre `y`.
57. `repeat x` es la lista infinita `[x, x, x, ...]`.
58. `replicate n x` es la lista formada por `n` veces el elemento `x`.
59. `reverse xs` es la inversa de la lista `xs`.
60. `round x` es el redondeo de `x` al entero más cercano.
61. `scanr f e xs` es la lista de los resultados de plegar `xs` por la derecha con `f` y `e`.
62. `show x` es la representación de `x` como cadena.
63. `signum x` es 1 si `x` es positivo, 0 si `x` es cero y -1 si `x` es negativo.
64. `snd p` es el segundo elemento del par `p`.
65. `splitAt n xs` es `(take n xs, drop n xs)`.
66. `sqrt x` es la raíz cuadrada de `x`.
67. `sum xs` es la suma de la lista numérica `xs`.
68. `tail xs` es la lista obtenida eliminando el primer elemento de `xs`.
69. `take n xs` es la lista de los `n` primeros elementos de `xs`.
70. `takeWhile p xs` es el mayor prefijo de `xs` cuyos elementos satisfacen el predicado `p`.
71. `uncurry f` es la versión cartesiana de la función `f`.
72. `until p f x` aplica `f` a `x` hasta que se verifique `p`.
73. `zip xs ys` es la lista de pares formado por los correspondientes elementos de `xs` e `ys`.
74. `zipWith f xs ys` se obtiene aplicando `f` a los correspondientes elementos de `xs` e `ys`.

A.1. Resumen de funciones sobre TAD en Haskell

A.1.1. Polinomios

1. `polCero` es el polinomio cero.
2. `(esPolCero p)` se verifica si `p` es el polinomio cero.
3. `(consPol n b p)` es el polinomio $bx^n + p$.
4. `(grado p)` es el grado del polinomio `p`.

5. `(coefLider p)` es el coeficiente líder del polinomio `p`.
6. `(restoPol p)` es el resto del polinomio `p`.

A.1.2. Vectores y matrices (`Data.Array`)

1. `(range m n)` es la lista de los índices del `m` al `n`.
2. `(index (m,n) i)` es el ordinal del índice `i` en `(m,n)`.
3. `(inRange (m,n) i)` se verifica si el índice `i` está dentro del rango limitado por `m` y `n`.
4. `(rangeSize (m,n))` es el número de elementos en el rango limitado por `m` y `n`.
5. `(array (1,n) [(i, f i) | i <- [1..n]])` es el vector de dimensión `n` cuyo elemento `i`–ésimo es `f i`.
6. `(array ((1,1),(m,n)) [((i,j), f i j) | i <- [1..m], j <- [1..n]])` es la matriz de dimensión `m.n` cuyo elemento `(i,j)`–ésimo es `f i j`.
7. `(array (m,n) ivs)` es la tabla de índices en el rango limitado por `m` y `n` definida por la lista de asociación `ivs` (cuyos elementos son pares de la forma (índice, valor)).
8. `(t ! i)` es el valor del índice `i` en la tabla `t`.
9. `(bounds t)` es el rango de la tabla `t`.
10. `(indices t)` es la lista de los índices de la tabla `t`.
11. `(elems t)` es la lista de los elementos de la tabla `t`.
12. `(assocs t)` es la lista de asociaciones de la tabla `t`.
13. `(t // ivs)` es la tabla `t` asignándole a los índices de la lista de asociación `ivs` sus correspondientes valores.
14. `(listArray (m,n) vs)` es la tabla cuyo rango es `(m,n)` y cuya lista de valores es `vs`.
15. `(accumArray f v (m,n) ivs)` es la tabla de rango `(m,n)` tal que el valor del índice `i` se obtiene acumulando la aplicación de la función `f` al valor inicial `v` y a los valores de la lista de asociación `ivs` cuyo índice es `i`.

A.1.3. Tablas

1. `(tabla ivs)` es la tabla correspondiente a la lista de asociación `ivs` (que es una lista de pares formados por los índices y los valores).
2. `(valor t i)` es el valor del índice `i` en la tabla `t`.
3. `(modifica (i,v) t)` es la tabla obtenida modificando en la tabla `t` el valor de `i` por `v`.

A.1.4. Grafos

1. `(creaGrafo d cs as)` es un grafo (dirigido o no, según el valor de o), con el par de cotas cs y listas de aristas as (cada arista es un trío formado por los dos vértices y su peso).
2. `(dirigido g)` se verifica si `g` es dirigido.
3. `(nodos g)` es la lista de todos los nodos del grafo `g`.
4. `(aristas g)` es la lista de las aristas del grafo `g`.
5. `(adyacentes g v)` es la lista de los vértices adyacentes al nodo `v` en el grafo `g`.
6. `(aristaEn g a)` se verifica si `a` es una arista del grafo `g`.
7. `(peso v1 v2 g)` es el peso de la arista que une los vértices `v1` y `v2` en el grafo `g`.

Apéndice B

Método de Pólya para la resolución de problemas

B.1. Método de Pólya para la resolución de problemas matemáticos

Para resolver un problema se necesita:

Paso 1: Entender el problema

- ¿Cuál es la incógnita?, ¿Cuáles son los datos?
- ¿Cuál es la condición? ¿Es la condición suficiente para determinar la incógnita? ¿Es insuficiente? ¿Redundante? ¿Contradictoria?

Paso 2: Configurar un plan

- ¿Te has encontrado con un problema semejante? ¿O has visto el mismo problema planteado en forma ligeramente diferente?
- ¿Conoces algún problema relacionado con éste? ¿Conoces algún teorema que te pueda ser útil? Mira atentamente la incógnita y trata de recordar un problema que sea familiar y que tenga la misma incógnita o una incógnita similar.
- He aquí un problema relacionado al tuyo y que ya has resuelto ya. ¿Puedes utilizarlo? ¿Puedes utilizar su resultado? ¿Puedes emplear su método? ¿Te hace falta introducir algún elemento auxiliar a fin de poder utilizarlo?

- ¿Puedes enunciar al problema de otra forma? ¿Puedes plantearlo en forma diferente nuevamente? Recurre a las definiciones.
- Si no puedes resolver el problema propuesto, trata de resolver primero algún problema similar. ¿Puedes imaginarte un problema análogo un tanto más accesible? ¿Un problema más general? ¿Un problema más particular? ¿Un problema análogo? ¿Puede resolver una parte del problema? Considera sólo una parte de la condición; descarta la otra parte; ¿en qué medida la incógnita queda ahora determinada? ¿En qué forma puede variar? ¿Puedes deducir algún elemento útil de los datos? ¿Puedes pensar en algunos otros datos apropiados para determinar la incógnita? ¿Puedes cambiar la incógnita? ¿Puedes cambiar la incógnita o los datos, o ambos si es necesario, de tal forma que estén más cercanos entre sí?
- ¿Has empleado todos los datos? ¿Has empleado toda la condición? ¿Has considerado todas las nociones esenciales concernientes al problema?

Paso 3: Ejecutar el plan

- Al ejercutar tu plan de la solución, comprueba cada uno de los pasos
- ¿Puedes ver claramente que el paso es correcto? ¿Puedes demostrarlo?

Paso 4: Examinar la solución obtenida

- ¿Puedes verificar el resultado? ¿Puedes el razonamiento?
- ¿Puedes obtener el resultado en forma diferente? ¿Puedes verlo de golpe? ¿Puedes emplear el resultado o el método en algún otro problema?

G. Polya "Cómo plantear y resolver problemas" (Ed. Trillas, 1978) p. 19

B.2. Método de Pólya para resolver problemas de programación

Para resolver un problema se necesita:

Paso 1: Entender el problema

- ¿Cuáles son las *argumentos*? ¿Cuál es el *resultado*? ¿Cuál es *nombre* de la función? ¿Cuál es su *tipo*?
- ¿Cuál es la *especificación* del problema? ¿Puede satisfacerse la especificación? ¿Es insuficiente? ¿Redundante? ¿Contradictoria? ¿Qué restricciones se suponen sobre los argumentos y el resultado?
- ¿Puedes descomponer el problema en partes? Puede ser útil dibujar diagramas con ejemplos de argumentos y resultados.

Paso 2: Diseñar el programa

- ¿Te has encontrado con un problema semejante? ¿O has visto el mismo problema planteado en forma ligeramente diferente?
- ¿Conoces algún problema *relacionado* con éste? ¿Conoces alguna función que te pueda ser útil? Mira atentamente el tipo y trata de recordar un problema que sea familiar y que tenga el mismo tipo o un tipo similar.
- ¿Conoces algún problema familiar con una *especificación* similar?
- He aquí un problema *relacionado* al tuyo y que ya has resuelto. ¿Puedes utilizarlo? ¿Puedes utilizar su resultado? ¿Puedes emplear su método? ¿Te hace falta introducir alguna función auxiliar a fin de poder utilizarlo?
- Si no puedes resolver el problema propuesto, trata de resolver primero algún problema similar. ¿Puedes imaginarte un problema análogo un tanto más *accesible*? ¿Un problema más *general*? ¿Un problema más *particular*? ¿Un problema *análogo*?
- ¿Puede resolver una *parte* del problema? ¿Puedes deducir algún elemento útil de los datos? ¿Puedes pensar en algunos otros datos apropiados para determinar la incógnita? ¿Puedes cambiar la incógnita? ¿Puedes cambiar la incógnita o los datos, o ambos si es necesario, de tal forma que estén más cercanos entre sí?
- ¿Has empleado todos los datos? ¿Has empleado todas las restricciones sobre los datos? ¿Has considerado todas los requisitos de la especificación?

Paso 3: Escribir el programa

- Al escribir el programa, comprueba cada uno de los pasos y funciones auxiliares.
- ¿Puedes ver claramente que cada paso o función auxiliar es correcta?
- Puedes escribir el programa en *etapas*. Piensas en los diferentes *casos* en los que se divide el problema; en particular, piensas en los diferentes casos para los datos. Puedes pensar en el cálculo de los casos independientemente y *unirlos* para obtener el resultado final
- Puedes pensar en la solución del problema descomponiéndolo en problemas con datos más simples y uniendo las soluciones parciales para obtener la solución del problema; esto es, por *recursión*.
- En su diseño se puede usar problemas más generales o más particulares. Escribe las soluciones de estos problemas; ellas puede servir como guía para la solución del problema original, o se pueden usar en su solución.
- ¿Puedes apoyarte en otros problemas que has resuelto? ¿Pueden usarse? ¿Pueden modificarse? ¿Pueden guiar la solución del problema original?

Paso 4: Examinar la solución obtenida

- ¿Puedes comprobar el funcionamiento del programa sobre una colección de argumentos?
- ¿Puedes comprobar propiedades del programa?
- ¿Puedes escribir el programa en una forma diferente?
- ¿Puedes emplear el programa o el método en algún otro programa?

Simon Thompson *How to program it*, basado en G. Polya *Cómo plantear y resolver problemas*.

Bibliografía

[1] J. A. Alonso and M. J. Hidalgo. Piensa en Haskell (Ejercicios de programación funcional con Haskell). Technical report, Univ. de Sevilla, 2012.

[2] R. Bird. *Introducción a la programación funcional con Haskell*. Prentice–Hall, 1999.

[3] H. C. Cunningham. Notes on functional programming with Haskell. Technical report, University of Mississippi, 2010.

[4] H. Daumé. Yet another Haskell tutorial. Technical report, University of Utah, 2006.

[5] A. Davie. *An introduction to functional programming systems using Haskell*. Cambridge University Press, 1992.

[6] K. Doets and J. van Eijck. *The Haskell road to logic, maths and programming*. King's College Publications, 2004.

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

[8] P. Hudak. *The Haskell school of expression: Learning functional programming through multimedia*. Cambridge University Press, 2000.

[9] P. Hudak. The Haskell school of music (From signals to symphonies). Technical report, Yale University, 2012.

[10] G. Hutton. *Programming in Haskell*. Cambridge University Press, 2007.

[11] B. O'Sullivan, D. Stewart, and J. Goerzen. *Real world Haskell*. O'Reilly, 2008.

[12] G. Pólya. *Cómo plantear y resolver problemas*. Editorial Trillas, 1965.

[13] F. Rabhi and G. Lapalme. *Algorithms: A functional programming approach*. Addison–Wesley, 1999.

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

[15] S. Thompson. *Haskell: The craft of functional programming*. Addison–Wesley, third edition, 2011.