INFORMATICA.
EXAMEN PARCIAL Febrero 2006. Haskell

Apellidos, Nombre

La puntuación total del examen es 10 puntos. Para aprobar es necesario un mínimo de 5 puntos.
NOTA: Para todas las funciones que definas en el examen, da además su tipo.

---

1) Sea el siguiente tipo para representar expresiones aritméticas con valores y variables enteras:

```
type NombreVar = String

data Expr = Const Int
          | Var NombreVar
          | Expr :+: Expr
          | Expr :-: Expr
          | Expr :*: Expr
          deriving Show
```

Así, la expresión matemática 1 + (*x* * *y*) queda representada como:

```
e1 :: Expr
e1 = Const 1 :+: (Var "x" :*: Var "y")
```

Una *asignación* es una correspondencia que da un valor entero a cada variable en una expresión. Para representar asignaciones, se definen los siguientes tipos:

```
data Asig = NombreVar :-> Int deriving Show
type Asignación = [Asig]
```

Por ejemplo:

```
a1 :: Asignación
a1 = [ "x" :-> 1, "y" :-> 3 ]
```

es una asignación que da el valor 1 a la variable *x* y el valor 3 a la variable *y*.

a) (*1 pto*) Define una función `evaluar` que dadas una `Expr` y una `Asignación`, devuelva el valor obtenido al evaluar la expresión bajo la asignación de variables dada. Por ejemplo:

```
evaluar e1 [ "x" :-> 1, "y" :-> 3 ]    ==>  4
evaluar e1 [ "x" :-> 2, "y" :-> 4 ]    ==>  9
```

El *dominio* de una variable es el conjunto de valores que puede tomar. Para representar dominios, se definen los siguientes tipos:

```
data Dom = NombreVar :==: [Int] deriving Show
type Dominio = [Dom]
```

Por ejemplo:

```
d1 :: Dominio
d1 = [ "x" :==: [1..3], "y" :==: [3..4] ]
```

indica que la variable *x* puede tomar valores entre 1 y 3 y que la variable *y* toma valores entre 3 y 4.

b) (*2 ptos*) Define una función `posiblesAsignaciones` que dado un `Dominio` devuelva una lista con todas las asignaciones de variables posibles según dicho dominio. Por ejemplo:

```
posiblesAsignaciones d1 ==>
 [["x" :-> 1, "y" :-> 3], ["x" :-> 2, "y" :-> 3],
  ["x" :-> 3, "y" :-> 3], ["x" :-> 1, "y" :-> 4],
  ["x" :-> 2, "y" :-> 4], ["x" :-> 3, "y" :-> 4] ]
```

2) Consideremos el siguiente tipo para representar matrices en Haskell como listas de filas:

```
type Fila = [Float]
type Matriz = [Fila]

m1 :: Matriz
m1 = [ [1,2,3]
     , [4,5,6]
     , [7,8,9]
     ]
```

a) (*1 pto*) Se dice que una matriz es cuadrada si el número de filas y columnas coincide. Escribe una función `esCuadrada`, que tome como parámetro una `Matriz` y compruebe si es cuadrada. Por ejemplo:

```
esCuadrada  m1      ==>  True
```

b) (*1.5 ptos*) Escribe una función `diagonalPrincipal` que tome como parámetro una `Matriz` cuadrada y devuelva una lista con los elementos de su diagonal principal. Por ejemplo:

```
diagonalPrincipal  m1    ==>  [1,5,9]
```

Si la matriz parámetro no fuese cuadrada, la función debe provocar un error.

c) (*1.5 ptos*) Escribe una función `noDiagonalPrincipal` que tome como parámetro una `Matriz` cuadrada y devuelva una lista con los elementos que no están en su diagonal principal. Por ejemplo:

```
noDiagonalPrincipal  m1 ==>  [2,3,4,6,7,8]
```

Si la matriz parámetro no fuese cuadrada, la función debe provocar un error.

d) (*1 pto*) La función predefinida `all :: (a->Bool) -> [a] -> Bool` toma una propiedad y una lista de elementos y devuelve `True` si todos los elementos de la lista cumplen la propiedad. Si al menos un elemento no la cumple, devuelve `False`. Por ejemplo:

```
all even [2,6,12]   ==>  True
all even [2,6,7,13] ==>  False
```

Define la función `all` **usando** `foldr`.

e) (*1 pto*) Se dice que una matriz cuadrada es diagonal si todos los elementos de la diagonal principal son distintos de cero y todos los que no están en la diagonal principal son cero. Escribe, **usando la función** `all`, una función `esDiagonal` que tome como parámetro una `Matriz` cuadrada y compruebe si es diagonal. Por ejemplo:

```
esDiagonal [ [1,0,0], [0,2,0], [0,0,3] ]  ==> True
```

Si la matriz parámetro no fuese cuadrada, la función debe provocar un error.

3) (*1 pto*) Calcula el tipo polimórfico de la función `g` y da una definición equivalente **usando una lista por comprensión**:

```
g x = map ($x)
```