Hoja de Ejercicios 1

Resuelve los siguientes ejercicios en Haskell (correspondientes a los temas 2 y 3 del temario de la asignatura).

Ejercicios – Primera parte

a) Implementar una función en Haskell que dados tres números enteros determine si están ordenados de menor a mayor.

b) Implementar una función en Haskell que dados tres números enteros los devuelva ordenados de menor a mayor.

c) Implementar en Haskell una función que reciba un número real y devuelva una tupla con su parte entera y sus dos primeros decimales (como número entero).

d) Crear una función que reciba el radio de una circunferencia y devuelva una 2-tupla con la longitud de la circunferencia y con el área del círculo. Emplea una definición local con la cláusula `where` para almacenar el valor de Pi (Nota: no se debe utilizar la función predefinida `pi`). A continuación crear una función con el mismo cometido empleando la definición local `let`.

e) Implementar la función predefinida de listas `concat`, que se llamará `concatenar`, utilizando la definición de listas por comprensión (no se puede utilizar recursividad).

f) Implementar una función que dado un número entero devuelva en una lista todos los factores de dicho número. Se debe utilizar la definición de listas por comprensión.

   En matemáticas, los **factores de un número** son los números enteros que pueden multiplicarse juntos para igualar ese número. O también se puede decir que los **factores de un número** son números enteros por el que un número es divisible.

g) Implementar una función que diga si un número es primo. Para ello se debe utilizar la función que calcula el número de factores de un número (ejercicio f).

   Nota: Si para resolver el ejercicio se deben comparar dos listas, se puede hacer con el operador de igualdad de listas (==). Por ejemplo:

```
> [1,2,3] == [1,2,3]
True
> [1,2,3] == [1,2]
False
```

h) Implementar una función que diga cuántos caracteres en mayúscula están contenidos en una frase dada. Se deberá utilizar la definición de listas por comprensión.

Ejercicios – Segunda parte

i) Implementar una función que dada una tupla de tres elementos, donde cada uno de ellos es a su vez una tupla de dos elementos de tipo `String` e `Int` respectivamente, retorne el primer elemento de cada tupla interna. Se deberá utilizar ajuste de patrones.

j) Implementar una función que devuelve `True` si la suma de los cuatro primeros elementos de una lista de números enteros es un valor menor a 10 y devolverá `False` en caso contrario. Se deberá utilizar ajuste de patrones.

k) Implementar una función que dado un carácter, que representa un punto cardinal, devuelva su descripción. Por ejemplo, dado `'N'` devuelva `"Norte"`.

l) Implementar una función que dada una frase retorne un mensaje donde se indique cuál es la primera y última letra de la frase original. Un ejemplo de aplicación de la función podría ser:

```
> procesarFrase "El perro de San Roque"
"La primera letra de la frase ''El perro de San Roque'' es 'E' y  la
ultima letra es 'e'"
```

Nota: No se permite el uso de recursividad. Se debe usar ajuste de patrones y se puede utilizar también patrones nombrados (para referirse a la cadena de entrada).

m) Implementar una función que dado un número entero devuelva mensajes indicando en qué rango de valores se encuentra dicho número (menor de 10, entre 10 y 20 o mayor de 20). Se debe utilizar definiciones locales.

Ejemplos de aplicación de la función son:

```
> clasificarValorEntrada 20
"El valor de entrada es mayor o igual a 10 y menor o igual a 20"

> clasificarValorEntrada 9
"El valor de entrada es menor que 10"

> clasificarValorEntrada 35
"El valor de entrada es mayor que 20"
```

Pista: La cadena “El valor de entrada” se repite constantemente, por ello una definición local tiene sentido para que sólo se defina una vez.

n) Implementar una función que dada una cadena de caracteres y un carácter, indique el número de apariciones del carácter en la cadena. No se debe utilizar recursividad, sí ajuste de patrones. Pista: utilizar la definición de listas por comprensión.

Ejemplos de aplicación de la función:

```
> contarApariciones "casa" 'c'
1

> contarApariciones "casa" 'a'
2

> contarApariciones "" 'c'
0
```

Hoja de Ejercicios 2

A continuación se muestran diferentes ejercicios para resolver con el lenguaje Haskell. Algunos ejercicios necesitan algoritmos recursivos para resolverlos (donde se puede utilizar la recursividad final y otros son para practicar con las expresiones lambda.

Ejercicios:

a) Implementa una función en Haskell que elimine de una lista de enteros aquellos números múltiplo de x.

```
> cribar [0,5,8,9,-9,6,0,85,-12,15] 2
[5,9,-9,85,15]
```

Se piden diferentes versiones de la misma función:
- Con definición de listas por comprensión
- Con recursividad no final
- Con recursividad final o de cola

b) Dada la siguiente definición de función

```
doble :: Int -> Int
doble x = x + x
```

¿Cómo cambiaría la definición utilizando expresiones lambda?

c) Se pide una función en Haskell que dada una lista de números enteros obtenga un número entero con el resultado de calcular el doble de cada uno de los elementos de la lista original y sumarlos todos. Se piden diferentes versiones de la misma función:
- Con recursividad no final
- Con recursividad final o de cola
- Utilizando expresiones lambda u orden superior (se puede hacer uso de la función predefinida de Haskell `map`).

```
> sumaDobles [2,3,4]
18

> sumaDobles [1,2,3]
12
```

d) Implementa una función que sume los cuadrados de los números pares contenidos en una lista de números enteros. Se piden dos versiones:
   a. Una versión que haga uso de las funciones de orden superior de listas `map` y `filter` para definir la nueva función.
   b. Una versión que utilice la definición de listas por comprensión.

e) Dada una lista de enteros, implementar una función para devolver tuplas formadas por los elementos (sin repetir) de la lista, junto con la primera posición en la que aparecen.

```
> primeraAparicion [1,5,6,0,2,6,4,78,9,41,-9,8,-9,12,45,0]
[(1,1),(5,2),(6,3),(0,4),(2,5),(4,7),(78,8),(9,9),(41,10),
                                        (-9,11),(8,12),(12,14),(45,15)]
```

f) Implementar en Haskell una función que calcule el número de secuencias de ceros que hay en una lista de números.

```
> ceros [0]                    > ceros[0,0]
1                              1
> ceros [0,1,0]                > ceros [0,0,1,5,0,4,0,0,0,5]
2                              3
```

g) Implementar una función en Haskell que reciba una lista de números enteros y devuelva dos listas: una con los elementos sin repetir y otra con los elementos que están repetidos.

```
> repeticiones [0,6,0,8,-2,-5,4,-2,6,98,71,2,0,5]
([8,-5,4,98,71,2,5],[0,6,-2])
```

h) Dada una lista de números enteros implementar una función que devuelva una lista con los *n* elementos mayores de la lista original.

```
> nmayores [8,4,-5,6,-1,0,2,6,-10,7] 4
[8,7,6,6]

> nmayores [8,4,-5,6,-1,0,2,6,-10,7] 7
[8,4,6,6,7,0,2]

> nmayores [8,4,-5,6,-1,0,2,6,-10,7] 11
[8,4,-5,6,-1,0,2,6,-10,7]
```

i) Implementa una función incluye en Haskell que reciba dos listas de números enteros y nos diga si la primera de las listas está contenida en la segunda. Se dice que una lista está contenida en otra si los elementos de la primera aparecen dentro de la segunda, en el mismo orden y de forma consecutiva.

```
> incluye [] [4,5]
True

> incluye [4,4,2] [5,4,4,5,4,4,2,9]
True

> incluye [4,4,2] [5,4,4,5,2,9]
False

> incluye [4,5] []
False
```

j) Dada una lista de enteros, se pide implementar una función que ordene dicha lista de menor a mayor utilizando un algoritmo de inserción. Dicho algoritmo de inserción consiste en recorrer la lista L, insertando cada elemento L[i] en el lugar correcto entre los elementos ya ordenados L[1] ,...,L[i-1].

```
> ordenar [2,3,1]
[1,2,3]

> ordenar [1,0,4,0,6,9]
[0,0,1,4,6,9]
```

k) Implementa una función polimórfica en Haskell que reciba 2 listas y vaya cogiendo un elemento de la primera y dos de la segunda, creando una lista final de ternas. En caso de que una de las dos listas se acabe, mostrará la lista de ternas construidas hasta ese momento.

```
> mezclarEnTernas [4,5,8,90] [0,5,6,-9,8,-1,9,52,22]
[(4,0,5),(5,6,-9),(8,8,-1),(90,9,52)]


> mezclarEnTernas [1,2,3] [5,6,7,8]
[(1,5,6),(2,7,8)]

> mezclarEnTernas [1,2,3,4,5] "atropellado"
[(1,'a','t'),(2,'r','o'),(3,'p','e'),(4,'l','l'),(5,'a','d')]

> mezclarEnTernas [True,False] [2.3,5.9,5.7]
[(True,2.3,5.9)]
```

l) Se pide una función polimórfica en Haskell que dado un elemento y una lista añada dicho elemento al final de la lista.

```
> alFinal 3 [1,2,6,7]
[1,2,6,7,3]
```

```
> alFinal True [False,False]
[False,False,True]

> alFinal 'k' "casita"
"casitak"
```

m) Mediante la programación de orden superior se pide implementar una de las funciones predefinidas en la librería estándar de Haskell: la función `zipWith`. Esta función recibe como parámetros una función y dos listas y une ambas listas aplicado la función entre los correspondientes parámetros.

```
> zipWith' max [6,3,2,1] [7,3,1,5]
[7,3,2,5]

> zipWith' (++) ["hola ", "ciao ", "hi "] ["pepe", "ciao", "peter"]
["hola pepe","ciao ciao","hi peter"]
ghci> zipWith' (*) (replicate 5 2) [1..]
[2,4,6,8,10]

> zipWith' (zipWith' (*)) [[1,2,3],[3,5,6]] [[3,2,2],[3,4,5]]
[[3,4,6],[9,20,30]]

> zipWith’ crearTupla [1,2,3] "casita"
[(1,'c'),(2,'a'),(3,'s')]
```

(Suponiendo que la función crearTupla tiene la siguiente definición:

```
crearTupla :: a-> b-> (a,b)
crearTupla x y = (x,y)
```

)

n) Define una función polimórfica que sea capaz de invertir los elementos de una lista. Se piden diferentes versiones:
   - Con recursividad no final
   - Con recursividad de cola o final
   - Utilizando la función de orden superior foldr

```
> reverse' [1,2,3]
[3,2,1]

> reverse' "casa"
"asac"
```

o) Define una función polimórfica que sea capaz de invertir los elementos de una lista de listas.

```
> reverse'' [[1,2,3],[3,4,5]]
[[5,4,3],[3,2,1]]
```

```
> reverse'' ["pepe", "casa", "patio"]
["oitap","asac","epep"]
```

p) Implementar la función predefinida de la librería estándar `flip`. Esta función lo que hace es recibir una función y devolver otra función que es idéntica a la función original, salvo que intercambia los dos primeros parámetros.

```
> flip' zip [1,2,3] "casa"
[('c',1),('a',2),('s',3)]

> flip' (+) 3 4
7

> flip' (++) "casa" "pollo"
"pollocasa"
```

q) Implementar la función polimórfica predefinida de la librería estándar `map`. Esta función lo que hace es recibir una función y una lista y devuelve la lista resultante de aplicar la función a cada elemento de la lista original.

```
> map (3*) [1,2,3]
[3,6,9]

> map doble [1,2,3]
[2,4,6]

> map not [True,False]
[False,True]
```

Hoja de Ejercicios 3

Listado de ejercicios para poner en práctica los conocimientos adquiridos sobre definición de tipos sinónimos y nuevos tipos, tipos recursivos y tipos recursivos polimórficos.

Ejercicios:

a) Se pide una función que dada una lista de racionales, donde cada racional se define como dos números enteros (numerador y denominador), y un número racional, devuelva otra lista con todos los racionales equivalentes al dado. Realiza dos versiones del ejercicio:
   1. Empleando `type`.
   2. Empleando `data`.

   Ejemplos de aplicación (si se utiliza `type`) serían:

```
> equivalentes [(2,4),(3,5),(4,8)] (1,2)
[(2.0,4.0),(4.0,8.0)]

> equivalents [(3,5)] (1,2)
[]
```

   Ejemplos de aplicación (si se utiliza `data`) serían:

```
> equivalentes[R(2,4),R(3,5),R(4,8)] (R(1,2))
[R (2.0,4.0),R (4.0,8.0)]
> equivalentes [R(3,5)] (R(1,2))
[]
```

b) Se pide varias funciones para hacer lo siguiente:
   1. Función que dado un punto de coordenadas y una dirección (Norte, Sur, Este u Oeste) mueva el punto hacia la dirección indicada. Un ejemplo de aplicación de la función sería:

```
> mover Este (3,4)
(4,4)

> mover Norte (3.5,9.2)
(3.5,10.2)
```

   2. Función que dados dos puntos de coordenadas indique cuál está más al sur. Ejemplos de aplicación de la función son:

```
> masAlSur (3,5) (4,6)
(3.0,5.0)

> masAlSur (4.5,-6.2) (4.5,-7)
(4.5,-7.0)
```

3. Función que calcule la distancia entre dos puntos:

```
> distancia (3,5) (6,7)
3.6055512
```

4. Función que dado un punto y una lista de direcciones, retorne el camino que forman todos los puntos después de cada movimiento sucesivo desde el punto original:

```
> camino (3.2,5.5) [Sur,Este,Este,Norte,Oeste]
[(3.2,4.5),(4.2,4.5),(5.2,4.5),(5.2,5.5),(4.2,5.5)]
```

c) Definir una función que dado un día de la semana, indique si éste es o no laborable. Para representar el día de la semana se deberá crear un nuevo tipo enumerado.

d) La empresa RealTimeSolutions, Inc. está trabajando en un controlador para una central domótica. El controlador recibe información de termostatos situados en diferentes habitaciones de la vivienda y basándose en esta información, activa o desactiva el aire acondicionado en cada una de las habitaciones. Los termostatos pueden enviar la información sobre la temperatura en grados Celsius o Fahrenheit. A su vez, los aparatos de aire acondicionado reciben dos tipos de órdenes: apagar y encender (on y off). Se pide:

1. Definir un tipo de datos para representar las temperaturas en ambos tipos de unidades.

2. Definir una función `convert` que dada una temperatura en grados Celsius la convierta a grados Fahrenheit y viceversa. (Conversión de C a F: f = c * 9/5 + 32; conversión de F a C: c = (f – 32) * 5/9.)

3. Definir un tipo de datos para representar las órdenes a los aparatos de a/a.

4. Definir una función `action` que dada una temperatura en cierta habitación determine la acción a realizar sobre el aparato de a/a de dicha habitación. El controlador debe encender el aparato si la temperatura excede de 28ºC. Ejemplos de aplicación:

```
> action(Celsius(25))
On
```

```
> action(Fahrenheit(83.5))
Off
```

e) Definir un tipo moneda para representar euros y dólares USA. Definir una función que convierta entre ambas monedas sabiendo que el factor de conversión de euros a dólares es 1.14.

f) Dada el siguiente tipo de datos recursivo que representa expresiones aritméticas:

```
data Expr = Valor Integer
          |Expr :+: Expr
          |Expr :-: Expr
          |Expr :*: Expr deriving Show
```

e.1) Se pide una función para calcular el valor de una expresión.
e.2) Se pide una función para calcular el número de constantes de una expresión.

Hoja de Ejercicios 4

Listado de ejercicios para poner en práctica los conocimientos adquiridos sobre definición de tipos sinónimos y nuevos tipos, tipos recursivos y tipos recursivos polimórficos. Y también sobre el manejo de clases de tipos en Haskell.

Ejercicios:

a) Se quiere ordenar los elementos de una lista (cuyos elementos son comparables) mediante el algoritmo del quicksort.

b) Se pide implementar una función que dada un número (de cualquier tipo que soporte la operación de división) y una lista de números del mismo tipo, divida a ese número por cada uno de los elementos contenidos en la lista y devuelva una lista con el resultado.
Ejemplos de aplicación de la función son:

```
> divisiones 5 [1,2,3]
[Just 5,Just 2,Just 1]

> divisiones 5 [1,2,3,0,9,10]
[Just 5,Just 2,Just 1,Nothing,Just 0,Just 0]
```

c) Dado un nuevo tipo de datos para representar un árbol binario de cualquier tipo, definido como sigue:

```
data Arbol a = AV | Rama (Arbol a) a (Arbol a)
```

Se pide definir una función que visualice el árbol por pantalla de una determinada forma: separando cada hijo izquierdo y derecho por "|", la raíz entre guiones y cada nivel diferente del árbol por "( )". Ejemplos de aplicación de la función sería los siguientes:

```
> mostrarArbol (Rama (Rama (Rama AV 60 AV) 8 AV) 5 (Rama AV 4 AV))
"((60)|-8-|())|-5-|(4)"

> mostrarArbol (Rama AV 5 (Rama AV 4 AV))
"()|-5-|(4)"
```

¿Sería equivalente a declarar el nuevo tipo de datos `Arbol` como una instancia de la clase `Show`?

```
data Arbol a = AV | Rama (Arbol a) a (Arbol a) deriving Show
```

d) Dado el siguiente tipo de datos que representa un árbol binario:

```
data Arbol a = AV | Rama (Arbol a) a (Arbol a) deriving Show
```

Se pide definir una función que calcule el espejo de un árbol.

Ejemplos de aplicación de la función serían:

```
> espejo (Rama (Rama (Rama AV 60 AV) 8 AV) 5 (Rama AV 4 AV))
Rama (Rama AV 4 AV) 5 (Rama AV 8 (Rama AV 60 AV))

> espejo (Rama AV 5 (Rama AV 4 AV))
Rama (Rama AV 4 AV) 5 AV
```

e) Se quiere poder mostrar por pantalla los datos de los estudiantes matriculados en una universidad que pertenezcan a alguna de las asociaciones de ésta (culturales, deportivas, de representación estudiantil, etc.). Para ello se deberán crear nuevos tipos de datos que representen:
   - Estudiante, de cada uno se debe disponer del nombre y titulación
   - Titulación, que pueden ser tres: Grado II, Grado II_ADE, Grado ADE
   - Lista de estudiantes matriculados
   - Lista de estudiantes que pertenecen a asociaciones

   Un ejemplo de aplicación de la función que se pide podría ser:

```
> mostrarAlumnosAsociaciones(listaMatriculados,listaAsociaciones)
"(Carlos Calle,GradoADE_II)(Irene Plaza,GradoADE)"
```

   Donde Carlos Calle e Irene Plaza son los únicos estudiantes matriculados que pertenecen a algún tipo de asociación en la universidad.

f) Se quiere poder representar una fecha de la siguiente forma: dd/mm/aaaa, para ello se deberá crear un nuevo tipo de datos en Haskell. Por ejemplo, si se crea un nuevo tipo de datos cuyo constructor de datos es Fecha, en el intérprete al poner fechas concretas nos devolvería la representación de la fecha que hayamos definido:

```
> Fecha 10 10 2013                > Fecha 24 12 2012
10/10/2013                        24/12/2012
```

g) Teniendo en cuenta el nuevo tipo de datos Fecha definido anteriormente, se pide una función que sea capaz de comparar dos fechas. Ejemplos de aplicación de la función serían:

```
> mismaFecha (Fecha 10 10 2013) (Fecha 10 10 2013)
True

> mismaFecha (Fecha 10 11 2013) (Fecha 10 10 2013)
False
```

h) Teniendo en cuenta la definición de la función qs del apartado (b) de este listado de ejercicios, se pide ordenar una lista de fechas mediante quicksort. Ejemplos de aplicación de la función serían:

```
> qs [(Fecha 10 10 2013), (Fecha 24 12 2012), (Fecha 10 09 2013), (Fecha
12 12 2013)]
[24/12/2012,10/9/2013,10/10/2013,12/12/2013]
```

i) Se pide crear una nueva clase de tipos, llamada `Coleccion`, para representar colecciones de datos de cualquier tipo, donde los tipos pertenecientes a esta clase tendrán el siguiente comportamiento:

```
esVacia: función para saber si la colección está vacía.
insertar: insertará un nuevo elemento en la colección.
primero: devolverá el primer elemento de la colección.
eliminar: eliminará un elemento de la colección.
size: devolverá el número de elementos de la colección.
```

Algunas de las funciones anteriores variarán su implementación en función del tipo de colección particular que sea instancia de la clase `Coleccion`. Por ello, se pide crear dos instancias diferentes de esta clase para los dos nuevos tipos de datos que se presentan a continuación:

```
data Pila a = Pil [a] deriving Show
data Cola a = Col [a] deriving Show
```

El primero de ellos representa una estructura de datos LIFO con elementos de tipo `a`. El segundo representa una estructura de datos FIFO de elementos de tipo `a`.

Ejemplos de aplicación de las funciones para ambos tipos de datos serían:

```
> insertar 10 (Col [1,2,3,4])
Col [1,2,3,4,10]

> insertar 10 (Pil [1,2,3,4])
Pil [1,2,3,4,10]

> primero (Col [1,2,3,4,10])
1

> primero (Pil [1,2,3,4,10])
10

> eliminar (Col [1,2,3,4,10])
Col [2,3,4,10]

> eliminar (Pil [1,2,3,4,10])
Pil [1,2,3,4]
```

Ejercicios E/S

1. Función que dado un listado de nombres lo muestre por pantalla en forma de tabla.

```
> escribeTabla ["pepe","caramelo","lluvia"]
1: pepe
2: caramelo
3: lluvia
```

2. Definir una función que sea capaz de leer dos líneas de la entrada estándar y las compare, escribiendo una cadena por pantalla que indique si son iguales o no.

```
> comparaCadenas
Introduce la primera linea: linea1
Introduce la segunda linea: hola
Las cadenas son diferentes
```

3. Definir una función que lea el contenido de un fichero de texto, lo procese invirtiendo todo el contenido y lo escriba de nuevo sobre el mismo fichero de entrada.

4. Definir una función que sea capaz de ir leyendo líneas de la entrada estándar y las va imprimiendo junto con el número de caracteres que tienen. Se irá ejecutando mientras no se encuentra una línea vacía. Un ejemplo del resultado de la ejecución de la función puede ser:

```
> leerLineas
Introduce una linea: hola
La linea tiene 4 caracteres
Introduce una linea: casita
La linea tiene 6 caracteres
Introduce una linea:
```

5. Definir una función que sea capaz de copiar el contenido de un fichero en otro.

```
> copiaFichero "fentrada.txt" "f2.txt"
```