Prácticas de Lenguajes, Tecnologías y Paradigmas de programación. Curso 2014-15

Parte II Programación Funcional

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Práctica 4: Tipos algebraicos y orden superior

Índice

1. Objetivo de la Práctica

En esta práctica se presenta:

- El manejo de listas y listas intensionales en GHCi, así como las funciones `map` y `filter`.
- El manejo de estructuras de datos mediante tipos construidos (tipos algebraicos) en Haskell.

Se han previsto 3 sesiones para resolver los ejercicios planteados en esta práctica.

2. Tipos e Inferencia de Tipos

En programación funcional, los valores que pueden obtenerse al evaluar expresiones están organizados por *tipos*. Cada tipo representa un conjunto de valores. Por ejemplo, el tipo (primitivo o básico) `Int` representa los valores `0`, `1`, `2`, `-1`, `-2`, etc. El tipo `Bool` representa los valores booleanos `True` y `False`. El tipo `Char` los caracteres alfanuméricos `'a'`, `'b'`, `'A'`, `'B'`, `'0'`, ... En programación funcional, como en la mayoría de lenguajes de programación, *toda expresión tiene asociado un tipo*. Esto se puede expresar explícitamente mediante el operador de tipificación '`::`'

```
42 :: Int
6*7 :: Int
doble (2+2) :: Int
```

Como ya se ha visto, se puede pedir al intérprete GHCi que informe acerca del tipo de cualquier expresión bien formada usando el comando de usuario "`:t`":

```
> :t 'a'
'a' :: Char


> :t "Hello"
"Hello" :: String
```

En programación funcional, los identificadores de las funciones son expresiones válidas gracias a la currificación y, como tales, tienen un tipo. Además, no es necesario que el programador proporcione información explícita de tipificación de las funciones que defina o de las expresiones utilizadas, ya que el intérprete lo *infiere* automáticamente a partir de las definiciones.

Se puede probar, por ejemplo:

```
> :t show
> :t (+)
> :t (*)
```

```
> :t (3 +)
> :t (* 2.0)
```

Nota adicional: Obsérvese el caso particular de la función binaria `(*)` que espera dos argumentos de un tipo numérico y devuelve un valor del mismo tipo numérico.

```
Prelude> :t (*)
(*) :: (Num a) => a -> a -> a
```

Sin embargo, la función unaria `(* 2.0)` espera sólo un argumento, además restringido al tipo `Float` o alguno parecido, y devuelve su doble, es decir, el número que resulta de multiplicar el argumento por la constante real 2.0.

```
Prelude> :t (* 2.0)
(* 2.0) :: (Fractional a) => a -> a
```

Obsérvese que esta función se ha obtenido como una simple aplicación parcial del operador de multiplicación `(*)`, que es binario, al caso particular de haber fijado su segundo argumento para que tome el valor de la constante real 2.0, lo cual es posible gracias a que esta función está definida de forma currificada.

3. Estrategia de Reducción

La estrategia de reducción en Haskell es *lazy* (perezosa). Esta estrategia reduce una expresión (parcialmente) sólo si realmente es necesario para calcular el resultado. Es decir, se reducen los argumentos sólo lo suficiente para poder aplicar algún paso de reducción en el símbolo de función más externo.

Gracias a esto, es posible trabajar con estructuras de datos infinitas. En lenguajes que usan la estrategia *eager* (impaciente), como son la mayoría de los lenguajes imperativos y los lenguajes funcionales más antiguos, este tipo de estructuras no pueden manipularse.

La función `repeat'` devuelve una lista infinita (similar a la función `repeat` del `Prelude`):

```
repeat' :: a -> [a]
repeat' x = xs where xs = x:xs
```

La llamada `repeat' 3` devuelve la lista infinita `[3,3,3,3,3,3,3,3,3,...`

Nota adicional: Se debe tener cuidado porque la llamada `repeat' 3` no termina e imprime una lista infinita del número 3 por la pantalla, teniendo que parar la ejecución con `^C`.

Una lista infinita generada por `repeat'` puede ser usada como argumento parcial por una función que tiene un resultado finito. La siguiente función, por ejemplo, toma un número finito de elementos de una lista:

```
take' :: Int -> [a] -> [a]
take' _ [] = []
take' n (x:t)
   | n<=0 = []
   | otherwise = x : take' (n - 1) t
```

Por ejemplo, la llamada `take' 3 (repeat' 4)` devuelve la lista `[4,4,4]`.

4. Las listas

4.1. El tipo `lista`

En programación funcional es posible emplear tipos estructurados cuyos valores están compuestos por objetos de otros tipos. Por ejemplo, el tipo lista `[a]` puede servir para aglutinar en una única estructura objetos del *mismo* tipo (denotado, en este caso, por la variable de tipo `a`, que puede instanciarse a cualquier tipo). En Haskell, las listas pueden especificarse encerrando sus elementos entre corchetes y separándolos con comas:

```
[1,2,3] :: [Int]
['a','b','c','d'] :: [Char]
[cos,log,(1.0 +)] :: [Float -> Float]
         -- Lista de funciones de
         -- reales en reales: coseno, logaritmo en base 2,
         -- incrementar una unidad un número real.
```

Sin embargo, las listas

```
[1,'a',2]
['a',log,3]
[cos,2,(*)]
```

no son válidas (¿por qué? ¿qué responde GHCi al preguntar por el tipo?).

La lista vacía se denota como `[]`. Cuando no es vacía, se puede descomponer usando una notación que separa el elemento inicial, de la lista que contiene los elementos restantes. Por ejemplo:

```
1:[2,3]   ó   1:2:[3]   ó   1:2:3:[]
```

```
'a':['b','c','d']  ó  'a':'b':['c','d'] ó 'a':'b':'c':'d':[]
cos:[log]   ó   cos:log:[]
```

Los tipos que incluyen variables de tipo en su definición (como el tipo lista) son tipos genéricos o *polimórficos*. Se pueden definir funciones sobre tipos polimórficos, que pueden emplearse sobre objetos de cualquier tipo que sea una instancia de los tipos polimórficos involucrados. Por ejemplo, la función (predefinida) `length` calcula la longitud de una lista:

```
> :t length
length :: [a] -> Int
```

La función `length` puede emplearse sobre listas de cualquier tipo base:

```
> length [1,2,3]
3
> length ['a','b','c','d']
4
> length [doble,cuadruple,fact]
3
```

El operador `(!!)` permite la indexación de listas. Se usa para obtener el elemento en una posición dada de una lista:

```
> :t (!!)
(!!) :: [a] -> Int -> a
```

La indexación puede utilizarse con listas de cualquier tipo base:

```
> [1,2,3] !! 2
3
>  ['a','b','c','d'] !! 0
'a'
```

Otra función muy útil para listas es `(++)`, que se usa para concatenar dos listas de cualquier tipo:

```
> [1,2,3] ++ [4,5,6]
[1,2,3,4,5,6]
> ['a','b','c','d'] ++ ['e','f']
"abcdef"
```

Ejercicio 1 (Resuelto) *Definir una función para calcular el valor binario correspondiente a un número entero no negativo x:*

```
decBin :: Int -> [Int]
```

Por ejemplo, la evaluación de la expresión:

```
> decBin 4
```

debe devolver la lista `[0,0,1]` *(empezando por el bit menos significativo).*

```
module DecBin where
 decBin :: Int -> [Int]
 decBin x = if x < 2 then [x]
            else (x `mod` 2) : decBin (x `div` 2)
```

Ejercicio 2 (Resuelto) *Definir una función para calcular el valor decimal correspondiente a un número en binario (representado como una lista de 1's y 0's):*

```
binDec :: [Int] -> Int
```

Por ejemplo, la evaluación de la expresión:

```
> binDec [0,1,1]
```

debe devolver el valor `6`.

```
module BinDec where
 binDec :: [Int] -> Int
 binDec (x:[]) = x
 binDec (x:y)  = x + binDec y * 2
```

Ejercicio 3 *Definir una función para calcular la lista de divisores de un número entero no negativo n:*

```
divisors :: Int -> [Int]
```

Por ejemplo, la evaluación de la expresión:

```
> divisors 24
```

debe devolver la lista `[1,2,3,4,6,8,12,24]`.

Ejercicio 4 *Definir una función para determinar si un entero pertenece a una lista de enteros:*

```
member :: Int -> [Int] -> Bool
```

Por ejemplo, la evaluación de la expresión:

```
> member 1 [1,2,3,4,8,9]
```

debe devolver el valor `True`. *Y la evaluación de la expresión:*

```
> member 0 [1,2,3,4,8,9]
```

debe devolver el valor `False`.

Ejercicio 5 *Definir una función para comprobar si un número es primo (sus divisores son 1 y el propio número) y una función para calcular la lista de los n primeros números primos:*

```
isPrime :: Int -> Bool
primes :: Int -> [Int]
```

Por ejemplo, la evaluación de la expresión:

```
> isPrime 2
```

debe devolver el valor `True`. *Y la evaluación de la expresión:*

```
> primes 5
```

debe devolver la lista `[1,2,3,5,7]`. *Se recuerda que Haskell permite obtener fácilmente una lista con los elementos iniciales de otra lista infinita (como, por ejemplo, la lista de todos los números primos).*

Ejercicio 6 *Definir una función para determinar cuántas veces aparece repetido un elemento en una lista de enteros:*

```
repeated :: Int-> [Int] -> Int
```

Por ejemplo, la evaluación de la expresión:

```
> repeated 2 [1,2,3,2,4,2]
```

debe devolver el valor `3`.

Ejercicio 7 *Definir una función para* concatenar *listas de listas, que tome como argumento una lista de listas y devuelva la concatenación de las listas consideradas:*

```
concat' :: [[a]] -> [a]
```

Por ejemplo, la evaluación de la expresión:

```
> concat' [[1,2],[3,4],[8,9]]
```

debe devolver la lista `[1,2,3,4,8,9]`.

Ejercicio 8 *Definir una función para* seleccionar *los elementos pares de una lista de enteros:*

```
selectEven :: [Int] -> [Int]
```

Por ejemplo, la evaluación de la expresión:

```
> selectEven [1,2,4,5,8,9,10]
```

devuelve la lista `[2,4,8,10]`.

Ejercicio 9 *Definir ahora una función para* seleccionar *los elementos que ocupan las "posiciones pares" de una lista de enteros (recuerda que las posiciones en una lista empiezan por el índice cero, siguiendo el funcionamiento del operador* `!!`*):*

```
selectEvenPos :: [Int] -> [Int]
```

Por ejemplo, la evaluación de la expresión:

```
> selectEvenPos [1,2,4,5,8,9,10]
```

devuelve la lista `[1,4,8,10]`.

Ejercicio 10 *Definir una función* `iSort` *para ordenar una lista en sentido ascendente. Para ello, definir antes una función* `ins` *que inserte correctamente un elemento en una lista ordenada (la ordenación se puede resolver*

recursivamente considerando sucesivas operaciones de inserción de los elementos a ordenar en la parte de la lista ya ordenada):

```
iSort :: [Int] -> [Int]
ins :: Int -> [Int] -> [Int]
```

Por ejemplo, la evaluación de la expresión:

```
> iSort [4,9,1,3,6,8,7,0]
```

devuelve la lista `[0,1,3,4,6,7,8,9]`*. Y la evaluación de la expresión:*

```
> ins 5 [0,1,3,4,6,7,8,9]
```

devuelve la lista `[0,1,3,4,5,6,7,8,9]`*.*

Las cadenas de caracteres vistas en la práctica anterior (por ejemplo, `"hello"`) son simples listas de caracteres escritas con una sintaxis especial, que omite las comillas simples. Así, `"hello"` es tan solo una sintaxis conveniente para la lista `['h','e','l','l','o']`. Toda operación genérica sobre listas es, por tanto, aplicable también a las cadenas.

4.2. Los rangos

Ya hemos visto los rangos en algunos ejemplos previos. La forma básica tiene la sintaxis `[first..last]` de modo que genera la lista de valores entre ambos (inclusive):

```
> [10..15]
[10,11,12,13,14,15]
```

La sintaxis de los rangos en Haskell permite las siguientes opciones:

- `[first..]`
- `[first,second..]`
- `[first..last]`
- `[first,second..last]`

Cuyo comportamiento se puede deducir de los siguientes ejemplos:

```
[0..] -> 0, 1, 2, 3, 4, ...
[0,10..] -> 0, 10, 20, 30, ...
[0,10..50] -> 0, 10, 20, 30, 40, 50
[10,10..] -> 10, 10, 10, 10, ...
[10,9..1] -> 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
[1,0..] -> 1, 0, -1, -2, -3, ...
[2,0..(-10)] -> 2, 0, -2, -4, -6, -8, -10
```

4.3. Las listas intensionales

Haskell proporciona una notación alternativa para las listas, las llamadas *listas intensionales* (notación que es útil también para describir cálculos que necesitan `map` y `filter`, tal como se verá en la siguiente sección). He aquí un ejemplo:

```
> [x * x | x <- [1..5], odd x]
[1,9,25]
```

La expresión se lee: la lista de los cuadrados de los números impares en el rango de 1 a 5.

Formalmente, una lista intensional es de la forma `[e|Q]`, donde `e` es una expresión y `Q` es un *cualificador*. Un cualificador es una secuencia, potencialmente vacía, de *generadores* y *guardas* separados por comas. Un generador toma la forma `x <- xs`, donde `x` es una variable o tupla de variables, y `xs` es una expresión de tipo lista. Una guarda es una expresión booleana. El cualificador `Q` puede ser vacío, en cuyo caso se escribe simplemente `[e]`. He aquí algunos ejemplos:

```
> [(a,b) | a <- [1..3], b <- [1..2]]
[(1,1),(1,2),(2,1),(2,2),(3,1),(3,2)]
>
> [(a,b) | b <- [1..2], a <- [1..3]]
[(1,1),(2,1),(3,1),(1,2),(2,2),(3,2)]
```

Los generadores posteriores pueden depender de variables introducidas por los precedentes, como en:

```
> [(i,j) | i <- [1..4], j <- [i+1..4]]
[(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]
```

Se pueden intercalar libremente generadores y guardas:

```
> [(i,j) | i <- [1..4], even i, j <- [i+1..4], odd j]
[(2,3)]
```

5. Las funciones `map` y `filter`

Se trata de dos funciones predefinidas útiles para operar con listas. La función `map` aplica una función a cada elemento de una lista. Es una típica función de las denominadas *de orden superior*, al aceptar, como primer argumento, no valores cualesquiera sino *funciones*. Por ejemplo:

```
> map square [9,3]
[81,9]
> map (<3) [1,2,3]
[True,True,False]
```

donde `square` es la función que calcula el cuadrado de un número entero. La definición de `map` es:

```
map          :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

Existen bastante leyes algebraicas útiles relacionadas con `map`. Dos hechos básicos son:

$$\texttt{map}\ id = id$$
$$\texttt{map}\ (f\ \cdot\ g) = (\texttt{map}\ f)\ \cdot\ (\texttt{map}\ g)$$

donde id es la función identidad y "$\cdot$" es la composición de funciones. La primera ecuación expresa que al aplicar la función identidad a todos los elementos de una lista, la lista queda sin modificar. Las dos apariciones de `id` en la primera ecuación tienen tipos diferentes: el de la izquierda es id `:: a -> a` y el de la derecha id `:: [a] -> [a]`. La segunda ecuación expresa que aplicar primero g a todo elemento de una lista y después aplicar f a todo elemento de la lista resultado, da el mismo resultado que aplicar $f\ \cdot\ g$ a la lista original. Leída de derecha a izquierda, esta ley permite reemplazar dos recorridos de una lista por uno solo, con la correspondiente ganancia en eficiencia, por lo que se aconseja usarla siempre que se pueda.

Ejercicio 11 *Definir, usando la función* `map`*, una función para* duplicar *todos los elementos de una lista de enteros:*

```
doubleAll :: [Int] -> [Int]
```

Por ejemplo, la evaluación de la expresión:

```
> doubleAll [1,2,4,5]
```

devuelve la lista `[2,4,8,10]`*.*

La función `filter` toma una función booleana `p` y una lista `xs` y devuelve la sublista de `xs` cuyos elementos satisfacen `p`. Por ejemplo:

```
> filter even [1,2,4,5,32]
[2,4,32]
```

La definición de `filter` es:

```
filter          :: (a -> Bool) -> [a] -> [a]
filter p []     = []
filter p (x:xs) = if p x then x:filter p xs else filter p xs
```

Ejercicio 12 *Expresar mediante listas intensionales las definiciones de las funciones* `map` *y* `filter`*. Nota: Se las puede llamar* `map'` *y* `filter'` *para evitar conflictos con las funciones predefinidas del* `Prelude`*.*

Ejercicio 13 *¿Qué computa la expresión misteriosa que aparece a continuación? (donde* `sum` *es la función predefinida que suma los elementos de una lista de enteros):*

```
> (sum . map square . filter even) [1..10]
```

6. Tipos algebraicos

6.1. Enumeraciones

En un lenguaje funcional como Haskell, el programador puede definir nuevos tipos con sus valores asociados, empleando los llamados tipos algebraicos. Por ejemplo, con la declaración:

```
data Color = Red | Green | Blue
```

se establece un nuevo tipo de datos `Color`. El tipo `Color` contiene sólo tres valores o datos denotados por los correspondientes constructores de datos constantes `Red`, `Green` y `Blue`. Recuérdese que, en Haskell, los identificadores que se refieren a tipos de datos y a constructores de datos siempre comienzan por una *letra mayúscula*.

6.2. Tipos Renombrados

También se pueden declarar renombramientos de tipos ya definidos, denominados tipos "sinónimos". Por ejemplo:

```
type Distance = Float
type Angle = Float
type Position = (Distance,Angle)
type Pairs a = (a,a)
type Coordinates = Pairs Distance
```

De hecho, el tipo `String` es un renombramiento, como ya se había mencionado antes:

```
type String = [Char]
```

y se le puede aplicar todas las operaciones sobre listas, por ejemplo las comparaciones de orden (orden lexicográfico):

```
> "hola" < "halo"
False
> "ho" < "hola"
True
```

A menudo se prefiere mostrar las cadenas sin que aparezcan las dobles comillas en la salida y donde los caracteres especiales, como salto de línea, etc., se impriman como el caracter real que representan. Haskell dispone de *órdenes* primitivas para imprimir cadenas, leer de ficheros, etc. Por ejemplo, usando la función `chr` que convierte un entero al carácter que éste representa:

```
> putStr ("Esta frase tiene un" ++ [chr 10] ++ "fin de linea")
Esta frase tiene un
fin de linea
```

Se recuerda la necesidad de importar el mdulo `Data.Char` para usar la función `chr`.

Ejercicio 14 *Define los siguientes tipos "sinónimos":*

```
type Person = String
type Book = String
type Database = [(Person,Book)]
```

El tipo `Database` *define una base de datos de una biblioteca como una lista de pares* `(Person,Book)` *donde* `Person` *es el nombre de la persona que tiene en préstamo el libro* `Book`*. Un ejemplo de base de datos es:*

```
exampleBase :: Database
exampleBase = [("Alicia","El nombre de la rosa"),("Juan",
 "La hija del canibal"),("Pepe","Odesa"),("Alicia",
 "La ciudad de las bestias")]
```

A partir de esta base de datos ejemplo se pueden definir funciones para obtener los libros que tiene en préstamo una persona dada, `obtain`*, para realizar un préstamo,* `borrow`*, y para realizar una devolución,* `return'`*. Por ejemplo, la función* `obtain` *se puede definir así:*

```
obtain :: Database -> Person -> [Book]
obtain dBase thisPerson
 = [book | (person,book) <- dBase, person == thisPerson]
```

que significa que la función devuelve la lista de todos los libros tales que hay un par `(person,book)` *en la base de datos y* `person` *es igual a la persona cuyos libros se está buscando. Por ejemplo, la evaluación de la expresión:*
`obtain exampleBase "Alicia"`
devuelve la lista: `["El nombre de la rosa","La ciudad de las bestias"]`

Completar el programa con las definiciones para las funciones `borrow` *y* `return'`*:*

```
borrow :: Database -> Book -> Person -> DataBase
return' :: Database -> (Person,Book) -> DataBase
```

6.3. Árboles

La declaración:

```
data TreeInt = Leaf Int | Branch TreeInt TreeInt
```

establece un nuevo tipo de datos `TreeInt` (donde `TreeInt` es, de nuevo, un constructor de tipo constante) que consta de un número infinito de valores definidos recursivamente con la ayuda de los símbolos constructores de datos

`Leaf` (unario) y `Branch` (binario), que toman como argumentos un número entero y dos árboles `TreeInt`, respectivamente.

Considérense algunos ejemplos de valores o datos de los tipos anteriores:

- `[Red,Green,Red]` es un valor de tipo `[Color]`
- `Branch (Leaf 0) (Branch (Leaf 0) (Leaf 1))` es un valor de tipo `TreeInt`.

Nada impide definir tipos genéricos empleando estos recursos expresivos. El tipo

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

es un tipo algebraico polimórfico para definir árboles de cualquier tipo de datos, de forma similar a como las listas admiten cualquier tipo de datos.

Por supuesto, también se pueden definir funciones sobre estos nuevos tipos de datos algebraicos definidos por el usuario. La función `numleaves` definida como sigue:

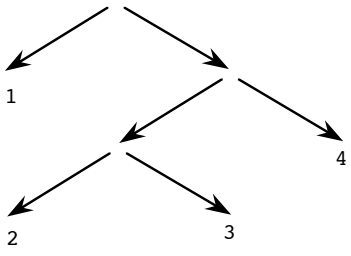
```
numleaves (Leaf x)     = 1
numleaves (Branch a b) = numleaves a + numleaves b
```

calcula el número de hojas de un árbol del tipo `Tree a`

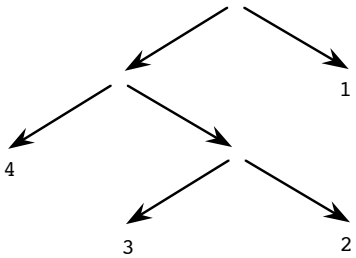
Ejercicio 15 *Escribir la definición de la función* `numleaves` *en un fichero y cargarlo en el intérprete* GHCi *para activar la definición de la función. Establecer la expresión de tipificación para la función* `numleaves` *y consultar la proporcionada por el intérprete.*

Ejercicio 16 *Definir una función que obtenga el árbol simétrico al que se le pasa como parámetro.*

```
symmetric :: Tree a -> Tree a
```



Un árbol de enteros...



...y su simétrico

Nota: Si pruebas la función `symmetric` con `ghci` verás que da un error:

```
> symmetric (Branch (Leaf 5) (Leaf 7))
<interactive>:5:1:
    No instance for (Show (Tree a0))
```

```
  arising from a use of 'print'
Possible fix: add an instance declaration for (Show (Tree a0))
In a stmt of an interactive GHCi command: print it
```

esto es debido a que no sabe cómo mostrar el resultado. El resultado se muestra con la función `show` (en cierta manera, como el `toString` de Java). Vamos a utilizar un mecanismo muy sencillo para utilizar un comportamiento por defecto para mostrar, con `show`, un tipo de datos algebraico. Basta con añadir `deriving Show` al finalizar la declaración de un tipo de datos algebraico:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a) deriving Show
```

Ejercicio 17 *Definir las funciones*

```
listToTree :: [a] -> Tree a
treeToList :: Tree a -> [a]
```

la primera de las cuales convierte una lista no vacía en un árbol, realizando la segunda de ellas la transformación contraria.

Considérese ahora la siguiente definición alternativa para el tipo de datos de los árboles binarios de enteros

```
data BinTreeInt  = Void | Node Int BinTreeInt BinTreeInt deriving Show
```

en el que los valores enteros se almacenan en los nodos y donde las hojas son subárboles que tienen tan solo raíz (denotados por el símbolo constructor de datos `Void`). A continuación, se muestran algunos ejemplos de árboles binarios de enteros:

```
let treeB1 = Void
let treeB2 = (Node 5) Void Void
let treeB3 = (Node 5)
              ((Node 3)(Node 1 Void Void)(Node 4 Void Void))
              ((Node 6) Void (Node 8 Void Void))
```

donde `treeB1` es un árbol vacío, `treeB2` es un árbol con un solo elemento, y `treeB3` es un árbol con el valor 5 en su nodo raíz, los valores 3, 1, 4 en los nodos de su rama izquierda, y los valores 6, 8 en los nodos de su rama derecha.

Se dice que un árbol binario está ordenado si los valores almacenados en el subárbol izquierdo de un nodo dado son siempre menores o iguales al valor en dicho nodo, mientras que los valores almacenados en su subárbol derecho son siempre mayores. Este tipo de árboles también recibe el nombre de árbol binario de búsqueda (*binary search tree*). Los anteriores ejemplos corresponden a árboles ordenados.

Sobre este tipo de datos resuélvanse los siguientes ejercicios.

Ejercicio 18 *Definir una función*

```
insTree :: Int ->  BinTreeInt -> BinTreeInt
```

para insertar un valor entero en su lugar en un árbol binario ordenado.

Ejercicio 19 *Dada una lista no ordenada de enteros definir una función*

```
creaTree :: [Int] ->  BinTreeInt
```

que construya un árbol binario ordenado a partir de la misma.

Ejercicio 20 *Definir una función*

```
treeElem :: Int ->  BinTreeInt -> Bool
```

que determine "de forma eficiente" si un valor entero pertenece o no a un árbol binario ordenado.

Ejercicios adicionales:

Ejercicio 21 *Considérese la anterior declaración de árbol binario de enteros:*

```
data BinTreeInt  = Void | Node Int BinTreeInt BinTreeInt deriving Show
```

Se quiere implementar una función `dupElem` *que devuelva un árbol con la misma estructura pero con todos sus valores duplicados.*

Considérese que se aplicara `dupElem` *a los árboles* `treeB1`, `treeB2` *y* `treeB3` *declarados anteriormente.*
La evaluación de la expresión:

```
> dupElem treeB1
```

devuelve `Void`*. La evaluación de la expresión:*

```
> dupElem treeB2
```

devuelve `Node 10 Void Void`*. Y la evaluación de la expresión:*

```
> dupElem treeB3
```

devuelve:

```
Node 10
(Node 6 (Node 2 Void Void) (Node 8 Void Void))
(Node 12 Void (Node 16 Void Void)).
```

Ejercicio 22 *Considérese la siguiente declaración:*

```
data Tree a = Branch a (Tree a) (Tree a) | Void deriving Show
```

Se quiere implementar una función `countProperty` *con la siguiente signatura:*

```
countProperty :: (a -> Bool) -> (Tree a) -> Int
```

que devuelva el número de elementos del árbol que cumplen la propiedad.
Por ejemplo, la evaluación de la expresión:

```
> countProperty (>9) (Branch 5 (Branch 12 Void Void) Void)
```

devuelve `1`*, y la evaluación de la expresión:*

```
> countProperty (>0) (Branch 5 (Branch 12 Void Void) Void)
```

devuelve `2`*.*

7. Evaluación

La asistencia a las sesiones de prácticas es obligatoria para aprobar la asignatura. La evaluación de esta segunda parte de prácticas se realizará mediante un examen individual en el laboratorio.