



Universidad  
de Huelva



Universidad de Huelva

GRADO EN INGENIERÍA INFORMÁTICA

# PRÁCTICA 4

DECLARACIÓN DE TIPOS DE DATO

Autor: Alberto Fernández Merchán

Asignatura: Modelos Avanzados de Computación

# Índice

<b>1. Introducción</b>	<b>2</b>
<b>2. Ejercicios</b>	<b>2</b>
2.1. DROPPRECIO X . . . . .	2
2.1.1. Código Fuente . . . . .	2
2.1.2. Descripción . . . . .	3
2.1.3. Funcionamiento . . . . .	3
2.2. GETLIST_DATE date criterio . . . . .	4
2.2.1. Código Fuente . . . . .	4
2.2.2. Descripción . . . . .	4
2.2.3. Funcionamiento . . . . .	5
2.3. RECORRE_ARBOL . . . . .	6
2.3.1. Código Fuente . . . . .	6
2.3.2. Descripción . . . . .	10
2.3.3. Funcionamiento . . . . .	11

## Índice de figuras

1. Código Fuente de la función dropPrecio . . . . .	2
2. Ejemplo de funcionamiento de la función dropPrecio . . . . .	3
3. Ejemplo de funcionamiento de la función dropPrecio . . . . .	3
4. Código Fuente de la función getListDate . . . . .	4
5. Ejemplo de funcionamiento de la función getDateList (despues) . . . . .	5
6. Ejemplo de funcionamiento de la función getDateList (antes) . . . . .	5
7. Ejemplo de funcionamiento de la función getDateList (misma) . . . . .	5
8. Declaración y creación de ArbolBinario . . . . .	6
9. Recorridos en Profundidad: Inorden, Postorden y Preorden . . . . .	7
10. Funciones auxiliares que utilizará el recorrido en anchura . . . . .	8
11. Recorrido en anchura . . . . .	8
12. Función principal para recorrer árboles binarios . . . . .	9
13. Ejemplo de la función recorreArbol . . . . .	11

# 1. Introducción

En esta práctica se nos proponen tres ejercicios en los que tendremos que crear nuestras propias clases para resolver el problema que nos proponen.

## 2. Ejercicios

### 2.1. DROPPRECIO X

#### 2.1.1. Código Fuente

```
----- Declaración de los tipos de dato -----
type ListaIngredientes = [String]
type Precio = Float
data Pizza = Pizza { ingredientes :: ListaIngredientes, precio::Precio
}deriving(Eq,Show)

----- Definición de las pizzas -----
margarita = Pizza {ingredientes = ["Tomate","Queso","Jamon York"], precio=10}
cuatroquesos = Pizza {ingredientes = ["Tomate","Roquefort","Gouda","Emmental","Mozzarella"], precio=12}
barbacoa = Pizza {ingredientes = ["Tomate", "Salsa BBQ", "Bacon", "Cebolla", "Terнера"], precio=16}
marinera = Pizza {ingredientes = ["Tomate", "Atun", "Anchoas","Aceitunas Negras"], precio=13}
campestre = Pizza {ingredientes = ["Tomate", "Champignones", "Pimiento", "Cebolla"], precio=9}
hawaiana = Pizza {ingredientes = ["Tomate", "Pina", "Maiz", "Jamon York"], precio= 12}

----- Creación de la Lista de Pizzas -----
listaPizzas = [margarita,cuatroquesos,barbacoa,marinera,campestre,hawaiana]

-- Obtiene la lista de pizzas con un precio superior a X
precioSuperiorA :: Precio -> [Pizza] -> [Pizza]
precioSuperiorA _ [] = []
precioSuperiorA x (cab:rest)
    | precio(cab) >= x = [cab]++(precioSuperiorA x rest)
    | otherwise = (precioSuperiorA x rest)

-- Obtiene la pizza con menor precio de una lista
getMenorPrecio :: [Pizza] -> Pizza
getMenorPrecio (cab:[]) = cab
getMenorPrecio (cab:(cab2:rest))
    | precio(cab) <= precio(cab2) = getMenorPrecio ([cab]++rest)
    | precio(cab2) < precio(cab) = getMenorPrecio ([cab2]++rest)

-- Obtiene el segmento que comienza con la pizza que tiene menor precio superior a X euros.
-- Llama a la función para que acepte un parámetro tipo lista de pizzas.
dropPrecio :: Precio -> [Pizza]
dropPrecio x = dropPrecioRec x listaPizzas

-- Obtiene el segmento que comienza con la pizza que tiene menor precio superior a X euros.
dropPrecioRec :: Precio -> [Pizza] -> [Pizza]
dropPrecioRec _ [] = []
dropPrecioRec x (cab:rest)
    | cab==getMenorPrecio(precioSuperiorA x ([cab]++rest)) = [cab]++rest
    | otherwise = dropPrecioRec x rest
```

Figura 1: Código Fuente de la función dropPrecio

### 2.1.2. Descripción

En este ejercicio deberemos devolver el segmento de la lista de *Pizzas* que comience con la pizza con el menor precio superior al pasado por parámetro. Como en el enunciado no se nos indica nada sobre que haya que pasarle por parámetro una lista de pizzas, he decidido utilizar una función auxiliar que utiliza la lista creada internamente.

Se ha hecho uso de dos funciones auxiliares que son:

- **precioSuperiorA**: permite calcular el subconjunto de pizzas cuyo precio sea superior o igual al pasado por parámetro.
- **getMenorPrecio**: permite calcular la pizza con menor precio de una lista pasada por parámetro.

La función recursiva se ha construido de la siguiente forma:

- **Caso Base**: Si la lista es una lista vacía, entonces devuelve una lista vacía.
- **Caso General**: Si el primer elemento de la lista es igual al elemento que tiene el menor precio superior al pasado por parámetro, devuelve dicho elemento seguido del resto de la lista. En caso de que no sea el mismo, vuelve a llamar a la función con un tamaño de la lista menor.

### 2.1.3. Funcionamiento

```
--  
Main> dropPrecio 9  
[Pizza {ingredientes = ["Tomate","Champignones","Pimiento","Cebolla"], precio =  
9.0},Pizza {ingredientes = ["Tomate","Pina","Maiz","Jamon York"], precio =  
12.0}] :: [Pizza]
```

Figura 2: Ejemplo de funcionamiento de la función dropPrecio

```
Main> dropPrecio 15  
[Pizza {ingredientes = ["Tomate","Salsa BBQ","Bacon","Cebolla","Ternera"],  
precio = 16.0},Pizza {ingredientes = ["Tomate","Atun","Anchoas","Aceitunas  
Negras"], precio = 13.0},Pizza {ingredientes =  
["Tomate","Champignones","Pimiento","Cebolla"], precio = 9.0},Pizza  
{ingredientes = ["Tomate","Pina","Maiz","Jamon York"], precio = 12.0}] ::  
[Pizza]
```

Figura 3: Ejemplo de funcionamiento de la función dropPrecio

## 2.2. GETLIST\_DATE date criterio

### 2.2.1. Código Fuente

```
----- Declaración de los tipos de dato -----
data Fecha = Fecha {dia::Integer, mes::Integer, agno::Integer}deriving(Eq,Show)
data Persona = Persona {nombre::String, apellidos::String, nacimiento::Fecha}deriving(Show)
-----

----- Creacion de las Personas -----
persona1 = Persona {nombre="Alberto",apellidos="Fernandez Merchan",nacimiento=(Fecha 29 3 2001)}
persona2 = Persona {nombre="Alba",apellidos="Marquez Rodriguez",nacimiento=(Fecha 7 5 2001)}
persona3 = Persona {nombre="Laura",apellidos="Fernandez Merchan",nacimiento=(Fecha 13 8 1997)}
persona4 = Persona {nombre="Pepe",apellidos="Castilla Rodriguez",nacimiento=(Fecha 20 4 1975)}
persona5 = Persona {nombre="Antonio", apellidos="Fernandez Rodriguez",nacimiento=(Fecha 16 8 1950)}
-----

----- Lista de Personas -----
listapersonas = [persona5, persona4, persona3, persona1, persona2]
-----

----- Funciones -----

--- Funciones para comparar fechas ---
esMenorFecha :: Fecha -> Fecha -> Bool
esMenorFecha fecha1 fecha2 = ((agno(fecha1)*10000) + (mes(fecha1)*100) + dia(fecha1)) <
((agno(fecha2)*10000) + (mes(fecha2)*100) + dia(fecha2))

esMayorFecha :: Fecha -> Fecha -> Bool
esMayorFecha fecha1 fecha2 = ((agno(fecha1)*10000) + (mes(fecha1)*100) + dia(fecha1)) >
((agno(fecha2)*10000) + (mes(fecha2)*100) + dia(fecha2))
-----

--- Funcion para obtener las personas que han nacido en cierta fecha. ---
getListDate :: Fecha -> String -> [Persona]
getListDate fecha criterio = case criterio of
    "antes"   -> [x| x<-listapersonas, esMenorFecha (nacimiento x) fecha]
    "despues" -> [x| x<-listapersonas, esMayorFecha (nacimiento x) fecha]
    "misma"   -> [x| x<-listapersonas, (nacimiento x) == fecha]
    otherwise -> error "Criterio no reconocido"
```

Figura 4: Código Fuente de la función getListDate

### 2.2.2. Descripción

En este ejercicio se nos pide que consigamos el subconjunto de personas que hayan nacido **antes**, **después**, o en una fecha concreta. Para ello se ha definido el tipo *Fecha* y el tipo *Persona* y un par de funciones auxiliares para poder comparar las fechas:

- **Comparación de Fechas:** Para comparar las fechas he optado por sumar los dígitos para ahorrar comparaciones. En lugar de comparar año, mes y día podemos reducirlo a comparar dos enteros.

Para la función principal, he optado por utilizar las **listas intensionales** ya que son adecuadas para este tipo de problemas donde:

- El **dominio** es la lista de personas que hemos definido.
- Las **condiciones** serán las diferentes comparaciones entre fechas.

### 2.2.3. Funcionamiento

```
Main> getListDate (Fecha 1 1 2000) "despues"
[Persona {nombre = "Alberto", apellidos = "Fernandez Merchan", nacimiento = Fecha
{dia = 29, mes = 3, agno = 2001}},Persona {nombre = "Alba", apellidos = "Marquez
Rodriguez", nacimiento = Fecha {dia = 7, mes = 5, agno = 2001}}] :: [Persona]
```

Figura 5: Ejemplo de funcionamiento de la función getDateList (despues)

```
Main> getListDate (Fecha 1 1 2000) "antes"
[Persona {nombre = "Antonio", apellidos = "Fernandez Rodriguez", nacimiento = Fecha
{dia = 16, mes = 8, agno = 1950}},Persona {nombre = "Pepe", apellidos = "Castilla
Rodriguez", nacimiento = Fecha {dia = 20, mes = 4, agno = 1975}},Persona {nombre =
"Laura", apellidos = "Fernandez Merchan", nacimiento = Fecha {dia = 13, mes = 8,
agno = 1997}}] :: [Persona]
```

Figura 6: Ejemplo de funcionamiento de la función getDateList (antes)

```
Main> getListDate (Fecha 20 4 1975) "misma"
[Persona {nombre = "Pepe", apellidos = "Castilla Rodriguez", nacimiento = Fecha {dia
= 20, mes = 4, agno = 1975}}] :: [Persona]
```

Figura 7: Ejemplo de funcionamiento de la función getDateList (misma)

## 2.3. RECORRE\_ARBOL

### 2.3.1. Código Fuente

En primer lugar, mostraré como se define la clase *ArbolBinario* y cómo se crea un árbol binario:

```
----- Declaración del tipo Arbol Binario -----  
data ArbolBinario = Vacio | Hoja {valor::Int} | Nodo {valor::Int, izq::ArbolBinario, der::ArbolBinario}  
deriving(Eq,Show)  
-----  
----- Declaración del arbol -----  
arbol = Nodo{valor=0, izq=(Nodo{valor=1,  
    izq=(Nodo{valor=3,  
        izq=(Hoja{valor=7}),  
        der=Vacio}),  
    der=(Nodo{valor=4,  
        izq=(Hoja{valor=8}),  
        der=Vacio})}),  
der=(Nodo{valor=2,  
    izq=(Nodo{valor=5,  
        izq=(Hoja{valor=9}),  
        der=Vacio}),  
    der=(Nodo{valor=6,  
        izq=(Hoja{valor=10}),  
        der=(Hoja{valor=11})})})})}
```

Figura 8: Declaración y creación de ArbolBinario

Los recorridos en profundidad se declaran de la siguiente forma:

```
----- Funcion de recorrido Inorden-----
-- Devuelve una lista con los valores de los nodos en inorden (izquierda, actual, derecha) --
-----
recorridoInorden :: ArbolBinario -> [Int]
recorridoInorden Vacio = []
recorridoInorden (Hoja x) = [x]
recorridoInorden (Nodo valor izq der) = (recorridoInorden izq) ++ [valor] ++ (recorridoInorden der)
-----

----- Funcion de recorrido Postorden -----
-- Devuelve una lista con los valores de los nodos en postorden (izquierda, derecha, actual) --
-----
recorridoPostorden :: ArbolBinario -> [Int]
recorridoPostorden Vacio = []
recorridoPostorden (Hoja x) = [x]
recorridoPostorden (Nodo valor izq der) = (recorridoPostorden izq) ++ (recorridoPostorden der) ++
[valor]
-----

----- Funcion de recorrido Preorden -----
-- Devuelve una lista con los valores de los nodos en preorden (actual, izquierda, derecha) --
-----
recorridoPreorden :: ArbolBinario -> [Int]
recorridoPreorden Vacio = []
recorridoPreorden (Hoja x) = [x]
recorridoPreorden (Nodo valor izq der) = [valor] ++ (recorridoPreorden izq) ++ (recorridoPreorden der)
-----
```

Figura 9: Recorridos en Profundidad: Inorden, Postorden y Preorden



```

----- Funcion nivelArbol -----
-- Devuelve una lista con los valores de los --
-- nodos del nivel pasado por parametro a la función. --
-----

nivelArbol :: ArbolBinario -> Int -> [Int]
-- Si el árbol es vacío devuelve una lista vacía
nivelArbol Vacio _ = []
-- Si el nivel del árbol es el nivel 0, devuelve el valor del nodo
nivelArbol (Nodo valor izq der) 0 = [valor]
nivelArbol (Hoja x) 0 = [x]
-- En caso de que no sea nivel 0, devuelve una lista vacía si es una hoja
nivelArbol (Hoja x) nivel = []
-- y si no es una hoja, llama recursivamente con un nivel inferior.
nivelArbol (Nodo valor izq der) nivel = (nivelArbol izq (nivel-1))++(nivelArbol der (nivel-1))
-----

----- Funcion alturaArbol -----
-- Devuelve la altura del árbol --
-----

alturaArbol :: ArbolBinario -> Int
alturaArbol (Hoja _) = 1
alturaArbol (Nodo _ izq Vacio) = 1 + (alturaArbol izq)
alturaArbol (Nodo _ Vacio der) = 1 + (alturaArbol der)
alturaArbol (Nodo _ izq der) = 1 + (max (alturaArbol izq) (alturaArbol der))
-----

```

Figura 10: Funciones auxiliares que utilizará el recorrido en anchura

```

----- Funcion de recorrido en Anchura -----
-- Devuelve una lista con los valores de los nodos del árbol --
-- recorridos en anchura (de izquierda a derecha) --
-----

recorridoAnchura :: ArbolBinario -> [Int]
recorridoAnchura arbol = recorridoAnchuraRec arbol ((alturaArbol arbol)-1)
-----

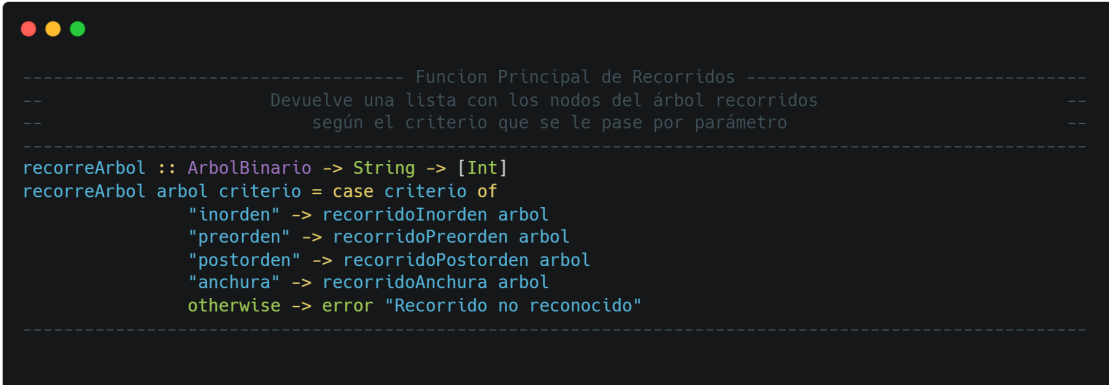
----- Funcion de recorrido en Anchura Recursiva -----
-- Devuelve una lista con los valores de los nodos del árbol --
-- recorridos en anchura (de izquierda a derecha) --
-- Concatena los valores de cada nivel del árbol desde el nivel 0 hasta el nivel (alturaArbol - 1) --
-----

recorridoAnchuraRec :: ArbolBinario -> Int -> [Int]
recorridoAnchuraRec arbol 0 = nivelArbol arbol 0
recorridoAnchuraRec arbol nivel = (recorridoAnchuraRec arbol (nivel-1))++(nivelArbol arbol nivel)
-----

```

Figura 11: Recorrido en anchura

Por último, la función principal que, según el criterio que se le pase por parámetro hará un recorrido u otro.

A screenshot of a code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. The code is written in Haskell and defines a function named `recorreArbol`. The function signature is `recorreArbol :: ArbolBinario -> String -> [Int]`. The function body uses a `case` expression to handle different traversal criteria: `"inorden"`, `"preorden"`, `"postorden"`, and `"anchura"`, each mapping to a corresponding traversal function. An `otherwise` clause returns an error message. The code is enclosed in a block of comments at the top.

```
----- Funcion Principal de Recorridos -----  
-- Devuelve una lista con los nodos del árbol recorridos --  
-- según el criterio que se le pase por parámetro --  
-----  
recorreArbol :: ArbolBinario -> String -> [Int]  
recorreArbol arbol criterio = case criterio of  
    "inorden" -> recorridoInorden arbol  
    "preorden" -> recorridoPreorden arbol  
    "postorden" -> recorridoPostorden arbol  
    "anchura" -> recorridoAnchura arbol  
    otherwise -> error "Recorrido no reconocido"
```

Figura 12: Función principal para recorrer árboles binarios

### 2.3.2. Descripción

A la hora de recorrer un árbol binario se puede hacer de diversas formas:

- **Profundidad:** Se trata de un recorrido en el que se visitan los nodos más profundos en primer lugar.
  - **Inorden:** Primero se recorre el nodo de la izquierda, en segundo lugar el nodo central y, por último, el nodo de la derecha.
  - **Postorden:** En primer lugar se recorre el nodo de la izquierda, después el de la derecha y, por último el central.
  - **Preorden:** En primer lugar se visita el nodo central, en segundo lugar el de la izquierda y, en último lugar, el de la derecha.
- **Anchura:** Es un recorrido que avanza de arriba hacia abajo y de izquierda a derecha.

Un árbol binario puede estar compuesto de 3 partes diferentes:

1. **Árbol Vacío:** Se considera el final del árbol. Se utiliza para parar la recursión.
2. **Hoja:** Es un nodo que se encuentra al final del árbol y no tiene hijos.
3. **Nodo:** Es un nodo intermedio que puede tener como máximo dos hijos (izquierda y derecha)

A partir de la definición anterior he creado la definición del tipo *ArbolBinario* (**Figura 8**).

Para realizar las funciones de recorrido en profundidad he aprovechado la definición recursiva de los árboles para codificarlas:

- **Caso base 1:** Si el nodo es *Vacío*: devuelve una lista vacía
- **Caso base 2:** Si el nodo es una *Hoja*: devuelve una lista con el valor de la hoja.
- **Caso general:** devuelve el recorrido correspondiente con los hijos del nodo (**Figura 9**).

Para el recorrido en anchura haremos uso de un par de funciones auxiliares (**Figura 10**):

- **nivelArbol:** Devuelve la lista de los nodos del nivel del árbol pasado por parámetro. Se calcula avanzando recursivamente un nivel hasta que el parámetro *nivel* sea 0. En ese caso se devolverá una lista con el valor del nodo central.  
En el caso de que el nodo sea una hoja, devolverá su valor y, si el nodo es *Vacío*, no devuelve nada.
- **alturaArbol:** Devuelve la altura del árbol pasado por parámetro. Se calcula como 1 más la altura máxima de los nodos hijos.

El recorrido en anchura se realizará llamando a una versión de la función recursiva (**Figura 11**) en la que va variando el nivel del árbol. La idea es concatenar la lista que nos proporciona la función *nivelArbol* para cada uno de los niveles del árbol. Comienza llamando a la función recursiva con un nivel igual a la altura del árbol menos uno. En la llamada recursiva se llama a la misma función con un nivel menos hasta llegar a 0 y se concatena el resultado con el del nivel superior.

Finalmente, la función principal (**Figura 12**) permite escoger el recorrido que queremos utilizar para visitar el árbol.

### 2.3.3. Funcionamiento

```
--  
Main> recorreArbol arbol "inorden"  
[7,3,1,8,4,0,9,5,2,10,6,11] :: [Int]  
Main> recorreArbol arbol "preorden"  
[0,1,3,7,4,8,2,5,9,6,10,11] :: [Int]  
Main> recorreArbol arbol "postorden"  
[7,3,8,4,1,9,5,10,11,6,2,0] :: [Int]  
Main> recorreArbol arbol "anchura"  
[0,1,2,3,4,5,6,7,8,9,10,11] :: [Int]  
--
```

Figura 13: Ejemplo de la función `recorreArbol`