Paradigmas de Programación

Práctica 11

Ejercicios:

1. El tipo de dato `'a bin_tree` definido a continuación puede servir para representar árboles binarios donde los nodos están etiquetados con valores de tipo `'a`:

```
type 'a bin_tree =
    Empty
  | Node of 'a bin_tree * 'a * 'a bin_tree
```

   Así, `Empty` podría representar el árbol vacío (sin nodos), y `Node (l, x, r)` representaría el árbol que tiene la raíz etiquetada con el valor `x`, y que tiene `l` y `r` como ramas izquierda y derecha, respectivamente.

   Se trata de implementar un módulo OCaml, de nombre `BinTree`, que reúna esta definición con una colección de valores (casi todos funciones) que permitan realizar las operaciones más habituales con estos árboles. En concreto, se pretende definir todos los valores descritos en la interfaz proporcionada `binTree.mli`.

   Para construir la implementación del módulo, copie la definición anterior en un archivo `binTree.ml` (éste sería el archivo de implementación del módulo `BinTree`) y añada todas las definiciones necesarias para satisfacer la interfaz proporcionada.

   Es una práctica habitual entre los programadores de OCaml, cuando se implementa un módulo que gira en torno a la definición de un tipo de dato, llamarle a ese tipo simplemente `t`, ya que el nombre del módulo debería ser ya bastante significativo. Para ello añada, después de la definición de `bin_tree`, la siguiente definición, que establece que `t` es un alias para `bin_tree`:

```
type 'a t =
  'a bin_tree
```

   Así, desde fuera del módulo, el constructor de tipos `bin_tree` se verá simplemente como `BinTree.t` en lugar de `BinTree.bin_tree`.

   Finalmente, complete la implementación con la definición de todos los valores que se mencionan en la interfaz del módulo. Para el desarrollo de esta implementación, seguramente sea cómodo utilizar el compilador interactivo `ocaml` (para hacer pruebas). Pero, una vez terminado, debe compilarse con la siguiente orden:

```
ocamlc -c binTree.mli binTree.ml
```

   Esto generará dos archivos (`binTree.cmi` y `binTree.cmo`) que son, respectivamente, la interfaz y la implementación compiladas del módulo `BinTree`.

   Si quiere utilizar este módulo desde el compilador interactivo `ocaml` debe cargarlo con el siguiente comando:

```
#load "binTree.cmo";;
```

   Una de las funciones más difíciles de definir de esta implementación es `breadth` (la que proporciona el recorrido en anchura de un árbol). Una posible definición sería la siguiente:

```
let breadth a =
  let rec aux = function
      [] -> []
    | Empty::t -> aux t
    | Node (l,x,r) :: t -> x :: aux (t @ [l;r]) (* ineficiente *)
  in aux [a]
```

Pero con esta definición, la función puede resultar muy ineficiente con algunos árboles (recuerde que la concatenación de listas es costosa).

Para comprobarlo puede utilizar la función `complete_tree` definida a continuación para generar un árbol binario "completo" con sus nodos numerados "en anchura":

```
let complete_tree n =
  let rec aux i =
    if i > n then Empty
    else Node (aux (2*i), i, aux (2*i+1))
  in aux 1
```

Pruebe la función `breadth` definida más arriba con árboles completos de, por ejemplo, 10.000 y 20.000 nodos. Seguramente podrá apreciar que ya tarda bastante. Cambie, por tanto, la definición de `breadth` para que no tenga este problema. Con una buena definición, `breadth` debería recorrer sin problema árboles mucho mayores (incluso de varios millones de elementos).

A continuación se muestra un ejemplo de uso del módulo `BinTree` en una sesión con el compilador interactivo `ocaml`:

```
$ ocaml
  OCaml version ...
# #load "binTree.cmo";;
# open BinTree;;

# let complete_tree n =
    let rec aux i =
      if i > n then empty
      else comb i (aux (2*i)) (aux (2*i+1))
    in aux 1;;
val complete_tree : int -> int BinTree.t = <fun>

# let t8 = complete_tree 8;;
val t8 : int BinTree.t = <abstr>

# size t8, height t8;;
- : int * int = (8, 4)

# breadth t8;;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8]

# let t8' = mirror t8;;
val t8' : int BinTree.t = <abstr>

# breadth t8';;
- : int list = [1; 3; 2; 7; 6; 5; 4; 8]
```

```
# preorder t8;;
- : int list = [1; 2; 4; 8; 5; 3; 6; 7]

# inorder t8;;
- : int list = [8; 4; 2; 5; 1; 6; 3; 7]

# postorder t8;;
- : int list = [8; 4; 5; 2; 6; 7; 3; 1]

# find_in_depth ((<) 3) t8;;
- : int = 4

# find_in_depth ((<) 3) t8';;
- : int = 7

# exists ((<) 7) t8;;
- : bool = true

# exists ((<) 8) t8;;
- : bool = false

# for_all ((<) 7) t8;;
- : bool = false

# for_all ((<) 0) t8;;
- : bool = true

# leaves t8;;
- : int list = [8; 5; 6; 7]

# leaves (right_branch t8);;
- : int list = [6; 7]

# leaves (right_branch t8');;
- : int list = [5; 8]

# let t8b = map (fun n -> n mod 2 = 0) t8;;
val t8b : bool BinTree.t = <abstr>

# breadth t8b;;
- : bool list = [false; true; false; true; false; true; false; true]

# let t8c = map (fun n -> if n mod 2 = 0 then 2 * n else n) t8;;
val t8c : int BinTree.t = <abstr>

# breadth t8c;;
- : int list = [1; 4; 3; 8; 5; 12; 7; 16]

# let tr = left_branch (left_branch t8);;
val tr : int BinTree.t = <abstr>

# breadth tr;;
- : int list = [4; 8]
```

2. El tipo de dato `'a st_bin_tree` definido a continuación puede servir para representar árboles estrictamente binarios[1] donde los nodos están etiquetados con valores de tipo `'a`:

```
type 'a st_bin_tree =
    Leaf of 'a
  | Node of 'a st_bin_tree * 'a * 'a st_bin_tree
```

Copie la definición anterior en un archivo con nombre `stBinTree.ml` y proceda como en el ejercicio anterior para implementar un módulo `StBinTree`, para trabajar con árboles estrictamente binarios, que satisfaga la interfaz descrita en el archivo `stBinTree.mli`. La descripción de cada uno de los valores declarados en esta interfaz se corresponde (salvo otra indicación) con la de sus homónimos en `binTree.mli`.

Este módulo debe compilar correctamente con la orden

```
ocamlc -c stBinTree.mli stBinTree.ml
```

Puesto que alguna de las funciones de este módulo precisa del módulo `BinTree`, para compilarlo necesitará tener presente el archivo `binTree.cmi` que produjo la compilación de `binTree.mli`.

Si quiere realizar pruebas en el compilador interactivo deberá cargar el módulo `BinTree` como se explicó en el enunciado del ejercicio anterior.

Tenga en cuenta que como los constructores del tipo de dato definido en la implementación del módulo `BinTree` no están declarados en su interfaz, no pueden ser utilizados fuera del propio módulo. Por tanto, no es posible utilizarlos, por ejemplo, en la definición de las funciones `to_bin` y `from_bin`.

A continuación se muestra un ejemplo de uso del módulo `StBinTree` en una sesión con el compilador interactivo `ocaml`:

```
$ ocaml
  OCaml version ...
# #load "binTree.cmo";;
# #load "stBinTree.cmo";;
# open StBinTree;;

# let complete_tree n = (* da el mismo valor para (2*i) y (2*i+1) *)
    let rec aux i =
        if 2*i > n then leaf_tree i
        else comb i (aux (2*i)) (aux (2*i+1))
   in aux 1;;
val complete_tree : int -> int StBinTree.t = <fun>

# let t9 = complete_tree 9;;
val t9 : int StBinTree.t =
  Node (Node (Node (Leaf 8, 4, Leaf 9), 2, Leaf 5), 1,
   Node (Leaf 6, 3, Leaf 7))

# size t9, height t9;;
- : int * int = (9, 4)
```

[1] En los árboles estrictamente binarios, todos los nodos que no sea hojas tienen exactamente dos nodos hijos.

```
# breadth t9;;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9]

# let t9' = mirror t9;;
val t9' : int StBinTree.t =
  Node (Node (Leaf 7, 3, Leaf 6), 1,
   Node (Leaf 5, 2, Node (Leaf 9, 4, Leaf 8)))

# breadth t9';;
- : int list = [1; 3; 2; 7; 6; 5; 4; 9; 8]

# preorder t9;;
- : int list = [1; 2; 4; 8; 9; 5; 3; 6; 7]

# inorder t9;;
- : int list = [8; 4; 9; 2; 5; 1; 6; 3; 7]

# postorder t9;;
- : int list = [8; 9; 4; 5; 2; 6; 7; 3; 1]

# find_in_depth ((<) 3) t9;;
- : int = 4

# find_in_depth ((<) 3) t9';;
- : int = 7

# exists ((<) 7) t9;;
- : bool = true

# exists ((<) 8) t9;;
- : bool = true

# for_all ((<) 7) t9;;
- : bool = false

# for_all ((<) 0) t9;;
- : bool = true

# leaves t9;;
- : int list = [8; 9; 5; 6; 7]

# leaves (right_branch t9);;
- : int list = [6; 7]

# leaves (right_branch t9');;
- : int list = [5; 9; 8]

# let t9b = map (fun n -> n mod 2 = 0) t9;;
val t9b : bool StBinTree.t =
  Node (Node (Node (Leaf true, true, Leaf false), true, Leaf false), false,
   Node (Leaf true, false, Leaf false))

# breadth t9b;;
- : bool list = [false; true; false; true; false; true; false; true; false]
```

```
# let t9c = map (fun n -> if n mod 2 = 0 then 2 * n else n) t9;;
val t9c : int StBinTree.t =
  Node (Node (Node (Leaf 16, 8, Leaf 9), 4, Leaf 5), 1,
   Node (Leaf 12, 3, Leaf 7))

# breadth t9c;;
- : int list = [1; 4; 3; 8; 5; 12; 7; 16; 9]

# let tr = left_branch (left_branch t9);;
val tr : int StBinTree.t = Node (Leaf 8, 4, Leaf 9)

# breadth tr;;
- : int list = [4; 8; 9]

# from_bin (to_bin t9) = t9;;
- : bool = true
```

3. El tipo de dato `'a g_tree` definido a continuación puede servir para representar árboles con nodos de cualquier aridad etiquetados con valores de tipo `'a`:

   ```
   type 'a g_tree =
       Gt of 'a * 'a g_tree list
   ```

   `Gt (x, lr)` representaría el árbol que tiene el valor `x` en la raíz y cuyas ramas son las de la lista `lr`.

   Copie la definición anterior en un archivo con nombre `gTree.ml` y proceda como en el ejercicio anterior para implementar un módulo `GTree` que satisfaga la interfaz descrita en el archivo `gTree.mli`. La descripción de cada uno de los valores declarados en esta interfaz se corresponde (salvo otra indicación) con la de sus homónimos en `binTree.mli`.

   Este módulo debe compilar correctamente con la orden

   ```
   ocamlc -c gTree.mli gTree.ml
   ```

   Puesto que alguna de las funciones de este módulo precisa de los módulos `BinTree` y `StBinTree`, para compilarlo necesitará tener presente los archivos `binTree.cmi` y `stBinTree.cmi`.

   Si quiere realizar pruebas en el compilador interactivo deberá cargar los módulos `BinTree` y `StBinTree` como se explicó en los enunciados anteriores.

   A continuación se muestra un ejemplo de uso del módulo `GTree` en una sesión con el compilador interactivo `ocaml`:

   ```
   $ ocaml
     OCaml version ...
   # #load "binTree.cmo";;
   # #load "stBinTree.cmo";;
   # #load "gTree.cmo";;
   # open GTree;;

   # let t5 = Gt (5, [leaf_tree 12]);;
   val t5 : int GTree.g_tree = Gt (5, [Gt (12, [])])
   ```

```
# let t3 = Gt (3, [t5; leaf_tree 6; leaf_tree 7]);;
val t3 : int GTree.g_tree =
  Gt (3, [Gt (5, [Gt (12, [])]); Gt (6, []); Gt (7, [])])

# let t4 = Gt (4, [leaf_tree 8; leaf_tree 9; leaf_tree 10; leaf_tree 11]);;
val t4 : int GTree.g_tree =
  Gt (4, [Gt (8, []); Gt (9, []); Gt (10, []); Gt (11, [])])

# let t2 = Gt (2, [t3; t4]);;
val t2 : int GTree.g_tree =
  Gt (2,
   [Gt (3, [Gt (5, [Gt (12, [])]); Gt (6, []); Gt (7, [])]);
    Gt (4, [Gt (8, []); Gt (9, []); Gt (10, []); Gt (11, [])])])

# let t1 = Gt (1, [t2]);;
val t1 : int GTree.g_tree =
  Gt (1,
   [Gt (2,
     [Gt (3, [Gt (5, [Gt (12, [])]); Gt (6, []); Gt (7, [])]);
      Gt (4, [Gt (8, []); Gt (9, []); Gt (10, []); Gt (11, [])])])])

# size t1, height t1;;
- : int * int = (12, 5)

# breadth t1;;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12]

# let t1' = mirror t1;;
val t1' : int GTree.t =
  Gt (1,
   [Gt (2,
     [Gt (4, [Gt (11, []); Gt (10, []); Gt (9, []); Gt (8, [])]);
      Gt (3, [Gt (7, []); Gt (6, []); Gt (5, [Gt (12, [])])])])])

# breadth t1';;
- : int list = [1; 2; 4; 3; 11; 10; 9; 8; 7; 6; 5; 12]

# preorder t1;;
- : int list = [1; 2; 3; 5; 12; 6; 7; 4; 8; 9; 10; 11]

# postorder t1;;
- : int list = [12; 5; 6; 7; 3; 8; 9; 10; 11; 4; 2; 1]

# leaves t1;;
- : int list = [12; 6; 7; 8; 9; 10; 11]

# leaves t1';;
- : int list = [11; 10; 9; 8; 7; 6; 12]

# find_in_depth ((<) 3) t1;;
- : int = 5

# find_in_depth ((<) 3) t1';;
- : int = 4
```

```
# find_in_depth ((<) 11) t1';;
- : int = 12

# find_in_depth ((<) 12) t1';;
Exception: Not_found.

# breadth_find ((<) 4) t1;;
- : int = 5

# breadth_find ((<) 4) t1';;
- : int = 11

# breadth_find ((<) 11) t1';;
- : int = 12

# breadth_find ((<) 12) t1';;
Exception: Not_found.

# exists ((<) 11) t1;;
- : bool = true

# exists ((<) 12) t1;;
- : bool = false

# for_all ((<) 7) t1;;
- : bool = false

# for_all ((<) 0) t1;;
- : bool = true

# for_all ((<) 4) t1;;
- : bool = false

# for_all ((<) 4) t5;;
- : bool = true

# let t1c = map (fun n -> char_of_int (96 + n)) t1;;
val t1c : char GTree.t =
  Gt ('a',
   [Gt ('b',
     [Gt ('c', [Gt ('e', [Gt ('l', [])]); Gt ('f', []); Gt ('g', [])]);
      Gt ('d', [Gt ('h', []); Gt ('i', []); Gt ('j', []); Gt ('k', [])])])])

# breadth t1c;;
- : char list =
  ['a'; 'b'; 'c'; 'd'; 'e'; 'f'; 'g'; 'h'; 'i'; 'j'; 'k'; 'l']

# let stl = StBinTree.leaf_tree;;
val stl : 'a -> 'a StBinTree.t = <fun>

# let comb = StBinTree.comb;;
val comb : 'a -> 'a StBinTree.t -> 'a StBinTree.t -> 'a StBinTree.t = <fun>
```

```
# let s1 = comb 1 (stl 2) (stl 3);;
val s1 : int StBinTree.t =
  StBinTree.Node (StBinTree.Leaf 2, 1, StBinTree.Leaf 3)

# let s4 = comb 4 (stl 5) (stl 6);;
val s4 : int StBinTree.t =
  StBinTree.Node (StBinTree.Leaf 5, 4, StBinTree.Leaf 6)

# let s = comb 0 s1 s4;;
val s : int StBinTree.t =
  StBinTree.Node (StBinTree.Node (StBinTree.Leaf 2, 1, StBinTree.Leaf 3), 0,
   StBinTree.Node (StBinTree.Leaf 5, 4, StBinTree.Leaf 6))

# breadth (from_bin (StBinTree.to_bin s));;
- : int list = [0; 1; 4; 2; 3; 5; 6]

# breadth (from_st_bin s);;
- : int list = [0; 1; 4; 2; 3; 5; 6]
```

4. Ejercicio opcional. Un árbol binario de búsqueda o BST (acrónimo del inglés para *Binary Search Tree*) es un tipo particular de árbol binario que cumple lo siguiente: el subárbol izquierdo de cualquier nodo (si no está vacío) contiene valores menores que el que contiene dicho nodo, y el subárbol derecho (si no está vacío) contiene valores mayores. Obviamente, esta definición asume que se pueden establecer relaciones de orden entre los valores de los nodos. De hecho, diferentes relaciones de orden podrían dar lugar a diferentes árboles binarios de búsqueda para un mismo conjunto de datos.

   Implemente, en un archivo de nombre `bst.ml`, las siguientes funciones:

   - `is_bst : ('a -> 'a -> bool) -> 'a BinTree.t -> bool`
     de tal forma que `is_bst ord t` indique si `t` es o no un árbol binario de búsqueda bien construido según la relación de orden `ord`.
   - `mem : ('a -> 'a -> bool) -> 'a -> 'a BinTree.t -> bool`
     de tal forma que `mem ord x t` indique si en `t` existe o no algún nodo etiquetado con el valor `x`.

     Esta función podría ser implementada utilizando `BinTree.exists` y un predicado adecuado. Pero en este caso no interesa realizar una búsqueda exhaustiva ni en profundidad ni en anchura, sino una búsqueda binaria mucho más eficiente que discrimine ramas en función de si los nodos que se van encontrando son "menores" o "mayores" (en términos de la relación de orden `ord`) que el valor `x` que se está buscando.

     De hecho, aunque no complicaremos tanto este ejercicio, existe un tipo especial de árboles de búsqueda que se denominan "balanceados", en el sentido de que almacenan los datos ordenados en un árbol con el menor número posible de niveles. En este tipo de árboles, las operaciones de búsqueda (y también las de inserción y borrado) tienen complejidad $\mathcal{O}(\log n)$ siendo n el número de nodos del árbol. Esto hace que los árboles binarios de búsqueda balanceados sean una estrutura muy eficiente para representar conjuntos de datos ordenados.

     Como hemos dicho, se sale del ámbito de este ejercicio mantener los árboles balanceados. Pero la función de búsqueda se programa de igual forma para árboles total o parcialmente balanceados. Lo peor que puede ocurrir es que la complejidad se transforme en $\mathcal{O}(n)$, siendo equivalente a realizar una búsqueda lineal en una lista.

- `add : ('a -> 'a -> bool) -> 'a -> 'a BinTree.t -> 'a BinTree.t`
  de tal forma que `add ord x t` devuelva el árbol resultante de insertar `x` en `t` en la posición que le corresponda según la relación de orden `ord`.

  Si ya existe algún nodo con valor `x`, la función devuelve el mismo árbol de entrada `t` (es decir, en el resultado no puede haber elementos repetidos). Y tal y como se puede deducir de todo lo dicho anteriormente, no es necesario que el árbol de salida quede balanceado.
- `remove : ('a -> 'a -> bool) -> 'a -> 'a BinTree.t -> 'a BinTree.t`
  de tal forma que `remove ord x t` devuelva el árbol resultante de eliminar `x` de `t` y dicho árbol siga cumpliendo `is_bst`. En este caso, el parámetro correspondiente a la relación de orden `ord` no sería estrictamente necesario, pero contribuye a que el elemento a eliminar sea localizado en el menor tiempo posible.

  Si no existe ningún nodo con valor `x`, la función devuelve el mismo árbol de entrada `t`. Y una vez más, no es necesario que el árbol de salida quede balanceado.

Dado que la implementación del tipo `BinTree.t` está oculta, a la hora de comprobar la corrección de estas implementaciones, puede hacer uso de la siguiente propiedad: el recorrido *inorder* de un árbol binario de búsqueda siempre produce una lista de valores ordenados de acuerdo a la relación de orden con la que fue creado dicho árbol.

Ejemplo de ejecución:

```
$ ocaml
  OCaml version ...
# #load "binTree.cmo";;
# open BinTree;;

# let t = comb 2 (leaf_tree 3) empty;;
val t : int BinTree.t = <abstr>

# is_bst (<) t, is_bst (>) t;;
- : bool * bool = (false, true)

# let from_list ord l =
    List.fold_left (fun t x -> add ord x t) empty l;;
val from_list : ('a -> 'a -> bool) -> 'a list -> 'a BinTree.t = <fun>

# let t = from_list (<) [15; 38; 8; 29; 22; 7; 3; 12];;
val t : int BinTree.t = <abstr>

# is_bst (<) t, inorder t;;
- : bool * int list = (true, [3; 7; 8; 12; 15; 22; 29; 38])

# mem (<) 7 t, mem (<) 17 t;;
- : bool * bool = (true, false)

# let t = remove (<) 8 t;;
val t : int BinTree.t = <abstr>

# is_bst (<) t, inorder t;;
- : bool * int list = (true, [3; 7; 12; 15; 22; 29; 38])
```

```
# let t =
    from_list (<) (List.init 1_000_000 (fun _ -> Random.int 100_000_000));;
val t : int BinTree.t = <abstr>

# let t = add (<) 100_000_000 t;;
val t : int BinTree.t = <abstr>

# is_bst (<) t;;
- : bool = true

# mem (<) 100_000_000 t;;
- : bool = true

# let t = remove (<) 100_000_000 t;;
val t : int BinTree.t = <abstr>

# is_bst (<) t;;
- : bool = true

# mem (<) 100_000_000 t;;
- : bool = false

# let l = inorder t;;
val l : int list = ...

# List.length l, l = List.sort compare l;;
- : int * bool = (995042, true)

# let crono f x =
    let t = Sys.time () in
    let _ = f x in
    Sys.time () -. t
val crono : ('a -> 'b) -> 'a -> float = <fun>

# let to_find = List.init 500 (fun _ -> Random.int 100_000_000);;
val to_find : int list = ...

# crono (List.filter (fun x -> List.mem x l)) to_find;;
- : float = 6.69775999999998817

# crono (List.filter (fun x -> mem (<) x t)) to_find;;
- : float = 0.00236799999998993371
```

Como se puede observar, la búsqueda sobre un árbol binario es mucho más rápida que sobre una lista, incluso aunque el árbol no esté balanceado.