

# Paradigmas de Programación

## Práctica 12

1. Descargue los archivos `g_tree.ml` y `g_tree.mli` e inspeccione su contenido. El archivo `g_tree.ml` debería contener la implementación de todas las funciones declaradas en el archivo `g_tree.mli`. Complete el archivo `g_tree.ml` con dichas implementaciones. Una vez completado, los archivos `g_tree.ml` y `g_tree.mli` constituirán, respectivamente, la implementación y la interfaz de un módulo que se denominará `G_tree`.

Si todo es correcto, debería poderse compilar este módulo sin problemas con la orden

```
ocamlc -c g_tree.mli g_tree.ml
```

Una vez compilado, el módulo puede cargarse desde el compilador interactivo `ocaml` con la directiva

```
#load "g_tree.cmo"
```

Si todo ha ido bien, debería poder reproducir la siguiente sesión de `ocaml` en su máquina:

```
# #load "g_tree.cmo";;
# open G_tree;;
# let t = Gt (2, [Gt (7, [Gt (2, []); Gt (10, []);
                    Gt (6, [Gt (5, []); Gt (11, [])])]);
              Gt (5, [Gt (9, [Gt (4, [])])])]);;
val t : int G_tree.g_tree =
  Gt (2,
    [Gt (7, [Gt (2, []); Gt (10, []); Gt (6, [Gt (5, []); Gt (11, [])])]);
     Gt (5, [Gt (9, [Gt (4, [])])])]
  )
# size t;;
- : int = 10
# height t;;
- : int = 4
# leaves t;;
- : int list = [2; 10; 5; 11; 4]
# mirror t;;
- : int G_tree.g_tree =
  Gt (2,
    [Gt (5, [Gt (9, [Gt (4, [])])]);
     Gt (7, [Gt (6, [Gt (11, []); Gt (5, [])]); Gt (10, []); Gt (2, [])])]
  )
# preorder t;;
- : int list = [2; 7; 2; 10; 6; 5; 11; 5; 9; 4]
# postorder t;;
- : int list = [2; 10; 5; 11; 6; 7; 4; 9; 5; 2]
```

2. (Ejercicio opcional) Considere ahora la función que, dado un árbol de tipo `'a g_tree`, devuelve la lista de nodos resultante de efectuar un recorrido por niveles sobre dicho árbol:

```
let rec breadth_first = function
  Gt (x, []) -> [x]
| Gt (x, (Gt (y, t2))::t1) -> x :: breadth_first (Gt (y, t1@t2));;
```

En un archivo de nombre `breadth_first.ml` escriba lo siguiente:

- Defina con nombre `breadth_first_t` una versión terminal de `breadth_first`.
- Defina un valor `t : int g_tree` tal que no sea posible calcular `breadth_first t`, pero sí sea posible calcular `breadth_first.t t`.

El archivo `breadth_first.ml` debe compilar con el archivo de interfaz proporcionado `breadth_first.mli` mediante la siguiente orden:

```
ocamlc -c g_tree.cmo breadth_first.mli breadth_first.ml
```

3. (Ejercicio opcional) Descargue los archivos `bin_tree.mli` y `bin_tree.ml` e inspeccione su contenido. El archivo `bin_tree.ml` debería contener la implementación de todas las funciones declaradas en el archivo `bin_tree.mli`. Complete el archivo `bin_tree.ml` con dichas implementaciones. Una vez completado, los archivos `bin_tree.ml` y `bin_tree.mli` constituirán, respectivamente, la implementación y la interfaz de un módulo que se denominará `bin_tree`.

Eso sí, a la hora de completar el archivo `bin_tree.ml`, tenga en cuenta lo siguiente. Observe estas dos funciones, que calculan, respectivamente, la suma de todas las etiquetas de un `int bin_tree` y el producto de todas las etiquetas de un `float bin_tree`:

```
let rec sum = function
  Empty -> 0
  | Node (x, l, r) -> x + (sum l) + (sum r);;

let rec prod = function
  Empty -> 1.0
  | Node (x, l, r) -> x *. (prod l) *. (prod r);;
```

La única diferencia entre estas dos funciones radica en el valor que devuelven cuando el árbol es vacío, y en la función que aplican cuando no lo es. Por eso es muy interesante considerar la implementación de una función

```
fold_tree : ('a -> 'b -> 'b -> 'b) -> 'b -> 'a bin_tree -> 'b
```

que permita generalizar cualquier operación de reducción sobre árboles binarios.

Así pues, en el archivo `bin_tree.ml`, las funciones `sum` y `prod` no deben implementarse de manera directa, tal y como hemos visto antes, sino que deben implementarse utilizando `fold_tree`. Y lo mismo ocurre con las funciones `size`, `inorder` y `mirror`. Todas ellas deben ser implementadas también mediante el uso de `fold_tree`.

Si todo es correcto, debería poderse compilar este módulo sin problemas con la orden

```
ocamlc -c bin_tree.mli bin_tree.ml
```

Una vez compilado, el módulo puede cargarse desde el compilador interactivo `ocaml` con la directiva

```
#load "bin_tree.cmo"
```

Si todo ha ido bien, debería poder reproducir la siguiente sesión de `ocaml` en su máquina:

```

# #load "bin_tree.cmo";;
# open Bin_tree;;
# let t = Node (3, Node (8, Empty, Empty),
                Node (2, Node (5, Empty, Empty),
                      Node (1, Empty, Empty)));;
val t : int Bin_tree.bin_tree =
  Node (3, Node (8, Empty, Empty),
        Node (2, Node (5, Empty, Empty), Node (1, Empty, Empty)))
# sum t;;
- : int = 19
# let rec map_tree f = function
  Empty -> Empty
  | Node (x,l,r) -> Node (f x, map_tree f l, map_tree f r)
  in prod (map_tree float t);;
- : float = 240.
# size t;;
- : int = 5
# inorder t;;
- : int list = [8; 3; 5; 2; 1]
# mirror t;;
- : int Bin_tree.bin_tree =
Node (3, Node (2, Node (1, Empty, Empty), Node (5, Empty, Empty)),
      Node (8, Empty, Empty))

```