Paradigmas de Programación

Práctica 9

1. Descargue todos los ficheros proporcionados con este mismo enunciado, y eche un vistazo a `bin_tree.mli` y `bin_tree.ml`. Estos ficheros no deben ser modificados. Se trata de un módulo para el manejo de árboles binarios definidos de la siguiente manera:

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

   Para utilizar este módulo dentro del compilador interactivo de OCaml, compílelo antes desde la línea de comandos con:

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

   Después ya puede cargarlo desde el compilador interactivo con la directiva:

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

   Y opcionalmente, para poder utilizar los nombres cortos de los valores y constructores definidos en el módulo (por ejemplo, `Empty` en lugar de `Bin_tree.Empty`), escriba también:

   ```
   open Bin_tree;;
   ```

   Sin embargo, para compilar un módulo o programa `p` que utilice el módulo `Bin_tree`, la orden de compilación desde la línea de comandos sería (probablemente sin referencia a la interfaz `p.mli`, si en vez de un módulo se trata de un programa ejecutable):

   ```
   ocamlc bin_tree.cmo p.mli p.ml
   ```

   Eso sí, tanto en `p.mli` como en `p.ml`, podría escribirse `open Bin_tree`.

   En ambos casos, una vez que el módulo ha sido abierto, podríamos definir árboles de prueba de esta manera:

   ```
   let t = Node (3, Node (8, Empty, Empty),
                    Node (2, Node (5, Empty, Empty),
                             Node (1, Empty, Empty)));;
   ```

   Observe ahora las dos siguientes funciones, que calculan la suma de todas las etiquetas de un `int bin_tree` y el producto de todas las etiquetas de un `float bin_tree`, respectivamente:

   ```
   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 tanto, realice las siguientes implementaciones en un fichero de nombre `ej91.ml`:

- Implemente una función

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

  que permita generalizar cualquier función de reducción sobre árboles binarios.
- Utilizando `fold_tree`, reimplemente `sum` y `prod`.
- Utilizando `fold_tree`, implemente también las funciones

  ```
  num_nodes : 'a bin_tree -> int
  in_order : 'a bin_tree -> 'a list
  mirror : 'a bin_tree -> 'a bin_tree
  ```

  que, dado un árbol binario, devuelven, respectivamente, su número de nodos, la lista de las etiquetas de todos sus nodos recorridos en "*in-order*", y el árbol binario correspondiente a su imagen especular.
- En el caso de la función `prod`, la operación recorre todo el árbol. Pero si algún nodo tuviera valor `0.0`, ya sabríamos que el resultado va a ser `0.0` y no sería necesario seguir multiplicando los valores de los restantes nodos. Así pues, reimplemente dicha función con nombre `prod2`, de tal forma que se incorpore el comportamiento que acabamos de describir. Sugerencia: active una excepción si aparece algún nodo con valor `0.0` y captúrela adecuadamente para devolver el resultado correcto.

El fichero `ej91.ml` debe compilar con el fichero de interfaz proporcionado `ej91.mli` mediante la siguiente orden:

```
ocamlc -c bin_tree.cmo ej91.mli ej91.ml
```

2. Considere el siguiente algoritmo de ordenación de listas:

   ```
   let insert_tree f x t = ... ;;

   let sort f l =
      in_order (List.fold_left (fun x a -> insert_tree f x a) Empty l) ;;
   ```

   Como se puede observar, el algoritmo inserta los elementos de la lista en un árbol binario ordenado (considerando la función `f` como criterio de ordenación) y posteriormente genera el recorrido en "*in-order*" de dicho árbol.

   Complete en un fichero de nombre `ej92.ml` la definición de la función

   ```
   insert_tree : ('a -> 'a -> bool) -> 'a -> 'a bin_tree -> 'a bin_tree
   ```

   para que este algoritmo de ordenación funcione.

   El fichero `ej92.ml` debe compilar con el fichero de interfaz proporcionado `ej92.mli` mediante la siguiente orden:

   ```
   ocamlc -c bin_tree.cmo ej92.mli ej92.ml
   ```

3. (Ejercicio opcional) Eche un vistazo a los ficheros `g_tree.mli` y `g_tree.ml` proporcionados con este mismo enunciado. Estos ficheros no deben ser modificados. Se trata de un módulo para el manejo de árboles generales definidos de la siguiente manera:

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

Este módulo puede ser compilado y utilizado de manera similar al módulo `Bin_tree` de los ejercicios anteriores. Una vez compilado, cargado y abierto el módulo, podríamos definir árboles generales de prueba como sigue:

```
let t = Gt (3, [ Gt (8, []);
                 Gt (2, [Gt (5, []);
                         Gt (4, []);
                         Gt (1, [])])]);;
```

Considere ahora la función que, dado un árbol de este tipo, 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 fichero de nombre `ej93.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 fichero `ej93.ml` debe compilar con el fichero de interfaz proporcionado `ej93.mli` mediante la siguiente orden:

```
ocamlc -c g_tree.cmo ej93.mli ej93.ml
```

4. (Ejercicio opcional) Eche un vistazo al fichero `st_tree.mli` proporcionado con este mismo enunciado. Este fichero no debe ser modificado. Se trata de la interfaz de un módulo `St_tree` que a través de un tipo de dato `'a st_tree` permite el manejo de árboles estrictamente binarios, en los cuales, de cada nodo que no sea hoja cuelgan exactamente dos ramas, y además no existen árboles estrictamente binarios vacíos (los más sencillos tienen un solo nodo que es raíz y hoja a la vez).

   Este módulo puede ser compilado y utilizado de manera similar a los módulos de los ejercicios anteriores. Pero dado que la implementación del tipo de dato `'a st_tree` no ha sido exportada en la interfaz del módulo, no es posible acceder a los constructores de dicho tipo de dato. Por lo tanto, la manera de construir valores de prueba de este tipo de dato sería únicamente a través de las funciones declaradas en la interfaz. De esta forma, una vez que el módulo haya sido compilado, cargado y abierto, podríamos escribir lo siguiente:

```
let t = comp 3 (single 8,
                comp 2 (single 5,
                        single 1));;
```

   Escriba en un fichero de nombre `ej94.ml` una función

```
breadth_first : 'a st_tree -> 'a list
```

   que calcule el recorrido por niveles de un árbol estrictamente binario.

   Recuerde que solo puede usar los valores definidos en la interfaz. En particular, le serán útiles las funciones `root` (que devuelve la raíz de un árbol) y `branches` (que devuelve las ramas izquierda y derecha de un árbol, o bien activa la excepción `No_branches` cuando se trata de un árbol sin ramas). Y por tanto quizás también le sea muy útil esta otra función, para saber si un árbol es sencillo, o si por el contrario tiene ramas:

```
let is_single t =
   try let _ = branches t in false
   with No_branches -> true;;
```

Escríbala en el fichero `ej94.ml` y tenga en cuenta que este fichero debe compilar con el fichero de interfaz proporcionado `ej94.mli` mediante la siguiente orden:

```
ocamlc -c st_tree.cmo ej94.mli ej94.ml
```