**Paradigmas de Programación**
**Práctica 7**

Se recomienda realizar la lectura completa de este enunciado antes de acometer la resolución de cada uno de los ejercicios propuestos. El espacio reservado para esta práctica en la planificación temporal de la asignatura es de dos semanas.

Los árboles "binarios llenos" o "estrictamente binarios" o "propiamente binarios" son árboles binarios en los que de todo nodo, que no sea hoja, "cuelgan exactamente dos ramas". No hay "árboles estrictamente binarios" vacíos; los más sencillos tienen un solo nodo que es raíz y hoja a la vez.

Los ejercicios 1, 2 y 3 tienen como objetivo la implementación en ocaml de un módulo `St_tree`, donde esté definido un tipo de dato `'a st_tree` que sirva para representar "árboles estrictamente binarios" con nodos etiquetados con valores de tipo `'a`. En dicho módulo estarán definidas también las funciones básicas para la creación y manipulación de "árboles estrictamente binarios".

## Ejercicio nº 1 (creación de la interfaz del módulo `St_tree`)

El módulo `St_tree` debe tener la siguiente interfaz:

```
type 'a st_tree

exception Ramas

val single : 'a -> 'a st_tree

val comp : 'a -> 'a st_tree * 'a st_tree -> 'a st_tree

val raiz : 'a st_tree -> 'a

val ramas : 'a st_tree -> 'a st_tree * 'a st_tree
```

Para ello, cree un fichero llamado `st_tree.mli` cuyo contenido sea este conjunto de líneas.

## Ejercicio nº 2 (creación de la implementación del módulo `St_tree`)

El módulo `St_tree` tendrá la siguiente implementación:

```
type 'a st_tree =
     S of 'a
   | C of 'a * 'a st_tree * 'a st_tree;;

exception Ramas;;

let single e = S e;;

let comp r (i,d) = C (r,i,d);;

let raiz = function S r | C (r,_,_) -> r;;

let ramas = function C (_,i,d) -> (i,d) | _ -> raise Ramas;;
```

Como es fácil deducir, la función `single` servirá para construir un árbol-hoja con una etiqueta dada.

La función `comp` servirá para construir un árbol a partir de una etiqueta para la raíz y de un par de árboles que serán sus ramas.

La función `raiz` devolverá el valor (etiqueta) de la raíz de un árbol.

La función `ramas` devolverá las ramas de un árbol. Si se aplica a un árbol que no tiene ramas (un árbol-hoja) se activará la excepción `Ramas`.

Para ello, cree un fichero llamado `st_tree.ml` cuyo contenido sea el conjunto de líneas anteriormente citadas.

**Ejercicio nº 3 (compilación del módulo `St_tree`)**

Compile los ficheros anteriores con la orden:

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

Compruebe que dicha orden genera correctamente los ficheros `st_tree.cmi` y `st_tree.cmo`.

A continuación, nos pondremos, no en el papel del desarrollador del módulo `St_tree`, sino en el papel de un usuario de dicho módulo (**que conocería la interfaz del módulo, pero no su implementación interna**) y ampliaremos el conjunto de funciones de manipulación de "árboles estrictamente binarios". Este objetivo quedará cubierto mediante la realización del ejercicio 4.

**Ejercicio nº 4 (ampliación del conjunto de funciones de manipulación de "árboles binarios llenos")**

Utilizando el módulo `St_tree`, pero asumiendo que sólo se conoce la interfaz del módulo, y no su implementación interna, defina en un fichero `st_tree_plus.ml` las siguientes funciones:

```
val is_single : 'a St_tree.st_tree -> bool (* identifica árboles-hoja *)

val izq : 'a St_tree.St_tree -> 'a St_tree.st_tree (* rama izquierda *)

val dch : 'a St_tree.st_tree -> 'a St_tree.st_tree (* rama derecha *)

val size : 'a St_tree.st_tree -> int (* número de nodos *)

val height : 'a St_tree.st_tree -> int (* altura *)

val preorder : 'a St_tree.st_tree -> 'a list

val postorder : 'a St_tree.st_tree -> 'a list

val inorder : 'a St_tree.st_tree -> 'a list

val leafs : 'a St_tree.st_tree -> 'a list (* lista de hojas *)

val mirror : 'a St_tree.st_tree -> 'a St_tree.st_tree (* imagen especular *)
```

```
val treemap : ('a -> 'b) -> 'a St_tree.st_tree -> 'b St_tree.st_tree
(* aplica una función a todos los nodos *)
```

(Nota: el archivo `st_tree_plus.ml` debería compilar sin errores con la orden `ocamlc -c st_tree_plus.ml`).

**Sugerencia nº 1**

Para utilizar el módulo `St_tree` puede cargarse el archivo `st_tree.cmo` desde el compilador interactivo de ocaml, con la directiva

```
#load "st_tree.cmo";;
```

El archivo `st_tree.cmi` debe estar presente en el mismo directorio.

Puede evitarse el tener que calificar los nombre de los valores con el nombre del módulo si se ejecuta la sentencia

```
open St_tree;;
```

**Sugerencia nº 2**

Para definir la función `is_single` puede interceptarse la excepción `Ramas` utilizando una construcción `try-with`.

Por ejemplo:

```
let is_single t =
    try let _ = ramas t in false
    with Ramas -> true;;
```

que también puede escribirse:

```
let is_single t =
    try ramas t ; false
    with Ramas -> true;;
```

ya que <exp1>; <exp2> es una expresión que se evalúa en ocaml evaluando primero la expresión <exp1> y a continuación la expresión <exp2> y cuyo valor es el de <exp2>.

**Ejercicio nº 5 (creación del módulo `Bin_tree`)**

Implemente ahora un módulo `Bin_tree` (es decir, escriba los correspondientes ficheros `bin_tree.mli` y `bin_tree.ml`) para representar cualquier tipo de árbol binario con nodos etiquetados (es decir, árboles en los que de cada nodo cuelgan a lo sumo dos ramas), basándose en la siguiente definición:

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

La interfaz de este módulo debe ser la siguiente:

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

exception Ramas

val empty : 'a bin_tree
val comp : 'a -> 'a bin_tree * 'a bin_tree -> 'a bin_tree
val raiz : 'a bin_tree -> 'a
val ramas : 'a bin_tree -> 'a bin_tree * 'a bin_tree
val is_empty : 'a bin_tree -> bool
val izq : 'a bin_tree -> 'a bin_tree
val dch : 'a bin_tree -> 'a bin_tree
val size : 'a bin_tree -> int
val height : 'a bin_tree -> int
val preorder : 'a bin_tree -> 'a list
val postorder : 'a bin_tree -> 'a list
val inorder : 'a bin_tree -> 'a list
val leafs : 'a bin_tree -> 'a list
val mirror : 'a bin_tree -> 'a bin_tree
val treemap : ('a -> 'b) -> 'a bin_tree -> 'b bin_tree
```

**Ejercicio nº 6 (compilación del módulo `Bin_tree`)**

Compile los ficheros anteriores con la orden:

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

Compruebe que dicha orden genera correctamente los ficheros `bin_tree.cmi` y `bin_tree.cmo`.

**Ejercicio nº 7 (conversión de árboles binarios)**

Utilizando los módulos `St_tree` y `Bin_tree`, defina en un nuevo fichero `tree_conversion.ml` las siguientes funciones:

```
val bin_tree_of_st_tree : 'a St_tree.st_tree -> Bin_tree.bin_tree
(* pasa de un st_tree al bin_tree correspondiente *)

val st_tree_of_bin_tree : 'a Bin_tree.bin_tree -> St_tree.st_tree
(* pasa de un bin_tree al st_tree correspondiente *)
```

Si en algún caso la conversión no es posible, debe activarse la excepción `Invalid_argument s`, donde `s` es el nombre de la función.

**Ejercicio nº 8 (opcional)**

Utilizando los módulos `St_tree` y `Bin_tree`, añada al fichero `tree_conversion.ml` las siguientes funciones:

```
val is_strict : 'a Bin_tree.bin_tree -> bool
```

```
val is_perfect : 'a Bin_tree.bin_tree -> bool
(* un árbol binario es perfecto si es estricto y todas sus hojas están en el
mismo nivel *)

val is_complet : 'a Bin_tree.bin_tree -> bool
(* un árbol es completo si todo nivel, excepto quizás el último, está lleno y
todos los nodos del último nivel están lo más a la izquierda posible *)
```

**Ejercicio nº 9 (opcional)**

Considere la siguiente definición de árbol general y escríbala en un nuevo fichero `g_tree.ml`:

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

Cualquier árbol general puede ser codificado como un árbol binario utilizando la idea descrita en la Wikipedia.

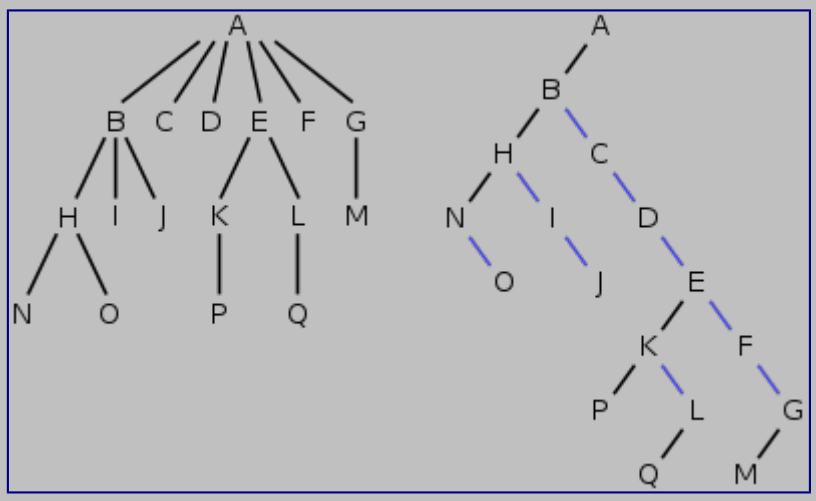
# Codificación de árboles n-arios como árboles binarios

Hay un mapeo uno a uno entre los árboles generales y árboles binarios, el cual en particular es usado en Lisp para representar árboles generales como árboles binarios. Cada nodo N ordenado en el árbol corresponde a un nodo N' en el árbol binario; el hijo de la izquierda de N' es el nodo correspondiente al primer hijo de N, y el hijo derecho de N' es el nodo correspondiente al siguiente hermano de N, es decir, el próximo nodo en orden entre los hijos de los padres de N.

Esta representación como árbol binario de un árbol general, se conoce a veces como un árbol binario primer hijo hermano, o un árbol doblemente encadenado.

Una manera de pensar acerca de esto es que los hijos de cada nodo estén en una lista enlazada, encadenados junto con el campo derecho, y el nodo solo tiene un puntero al comienzo o la cabeza de esta lista, a través de su campo izquierdo.

Por ejemplo, en el árbol de la izquierda, la A tiene 6 hijos (B, C, D, E, F, G). Puede ser convertido en el árbol binario de la derecha.




El árbol binario puede ser pensado como el árbol original inclinado hacia los lados, con los bordes negros izquierdos representando el primer hijo y los azules representado los siguientes hermanos.

Añada al fichero `g_tree.ml` las siguientes funciones:

```
val cod_as_bin : 'a g_tree -> 'a Bin_tree.bin_tree

val dec_from_bin : 'a Bin_tree.bin_tree -> 'a g_tree
```

para codificar y decodificar los árboles generales como binarios.

Independientemente de esta codificación, todo árbol binario puede verse como árbol general, y algunos árboles generales son también árboles binarios (o incluso estrictamente binarios). Complete entonces el módulo `Tree_conversion` con las funciones:

```
val g_tree_of_st_tree : 'a St_tree.st_tree -> G_tree.g_tree

val g_tree_of_bin_tree : 'a Bin_tree.bin_tree -> G_tree.g_tree

val st_tree_of_g_tree : 'a G_tree.g_tree -> St_tree.st_tree

val bin_tree_of_g_tree : 'a G_tree.g_tree -> Bin_tree.bin_tree
```