Recorridos de árboles binarios

1. Introducción

El recorrido de árboles se refiere al proceso de visitar de una manera sistemática, exactamente una vez, cada nodo en una estructura de datos de árbol (examinando y/o actualizando los datos en los nodos). Tales recorridos están clasificados por el orden en el cual son visitados los nodos. Los siguientes algoritmos son descritos para un árbol binario, pero también pueden ser generalizados a otros árboles.

Árbol binario

- **PreOrden:**. Para recorrer un árbol binario no vacío en *preorden*, hay que realizar las siguientes operaciones recursivamente en cada nodo, comenzando con el nodo de raíz:
  - Visite la raíz
  - Recorra en PreOrden el sub-árbol izquierdo
  - Recorra en PreOrden el sub-árbol derecho
- **InOrden:** Para recorrer un árbol binario no vacío en *inorden* (simétrico), hay que realizar las siguientes operaciones recursivamente en cada nodo:
  - Recorra en InOrden el sub-árbol izquierdo
  - Visite la raíz
  - Recorra en InOrden el sub-árbol derecho
- **PostOrden:** Para recorrer un árbol binario no vacío en *postorden*, hay que realizar las siguientes operaciones recursivamente en cada nodo:
  - Recorra en PostOrden el sub-árbol izquierdo
  - Recorra en PostOrden el sub-árbol derecho
  - Visite la raíz

  En general, la diferencia entre *preorden*, *inorden* y *postorden* es cuándo se recorre la raíz. En los tres, se recorre primero el sub-árbol izquierdo y luego el derecho.

2. Implementando los distintos recorridos

Consideremos la definición de árbol binario etiquetado ya vista:

```
- datatype 'etiqueta arbolbin =
            Vacio |
            Nodo of 'etiqueta * 'etiqueta arbolbin * 'etiqueta arbolbin;
datatype 'a arbolbin = Nodo of 'a * 'a arbolbin * 'a arbolbin | Vacio
```

Podemos definir las funciones `preOrden`, `inOrden` y `posOrden` que listan los elementos de un árbol binario en el orden especificado más arriba:

```
- fun preOrden(Vacio) = nil
|     preOrden(Nodo(a,izq,der)) = [a]@preOrden(izq)@preOrden(der);
val preOrden = fn : 'a arbolbin ->'a list

- fun inOrden(Vacio)=nil
|     inOrden(Nodo(a,izq,der)) = inOrden(izq)@[a]@inOrden(der);
```

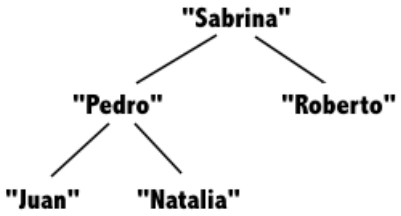
```
val inOrden = fn : 'a arbolbin ->'a list

- fun postOrden(Vacio)=nil
|     postOrden(Nodo(a,izq,der))=postOrden(izq)@postOrden(der)@[a];
val postOrden = fn : 'a arbolbin ->'a list
```

Si consideramos el ejemplo de tipo `string arbolbin` ya visto en apuntes anteriores:

```
- val ejArbol= Nodo("Sabrina",
                Nodo("Pedro",
                        Nodo("Juan", Vacio, Vacio),
                        Nodo("Natalia", Vacio, Vacio)
                ),
                Nodo("Roberto", Vacio, Vacio)
);
```

cuya representación gráfica es la siguiente:



y aplicamos las funciones recién definidas, obtenemos los siguientes resultados:

```
- preOrden(ejArbol);
val it = ["Sabrina","Pedro","Juan","Natalia","Roberto"] : string list

- inOrden(ejArbol);
val it = ["Juan","Pedro","Natalia","Sabrina","Roberto"] : string list

- postOrden(ejArbol);
val it = ["Juan","Natalia","Pedro","Roberto","Sabrina"] : string list
```