Ejercicio 1.- Se dice que un árbol binario es "zurdo" en uno de estos tres casos:

- si es el árbol vacío; o
- si es una hoja; o
- si sus hijos izquierdo y derecho son los dos "zurdos" y el hijo izquierdo tiene más elementos que el hijo derecho.

Crear las operaciones necesarias para determinar si un árbol binario es "zurdo".

Solución:
Operaciones
num_elementos: a_bin→natural {Operación auxiliar}
func num_elementos (a:a_bin) **dev** n:natural
    si vacio?(a) **entonces** n←0
    sino   n←1+num_elementos(izq(a))+num_elementos(der(a))
finfunc

zurdo: a_bin→bool
func zurdo (a:a_bin) **dev** b:bool
    si vacio?(a) V (vacio?(izq(a)) ∧ vacio?(der(a))) **entonces** b←T
        s**ino**
        b←(num_elementos(der(a))< num_elementos(izq(a)))
        ∧ zurdo (izq(a)) ∧ zurdo(der(a))

Ejercicio 2.- Extender el TAD árboles binarios de naturales, añadiendo operaciones para:

a) Obtener la suma de todos los elementos que sean números pares del árbol,
b) Obtener la imagen especular de un árbol (reflejo respecto al eje vertical),
c) Crear tres operaciones que generen una lista con los elementos del árbol recorrido en preorden, inorden y postorden,
d) Comprobar si el árbol está ordenado en inorden, usando para ello únicamente operaciones de árboles (en concreto, no puede utilizarse el apartado anterior).

<u>Solución:</u>

Los apartados a) y c) están resueltos en clase.

Operaciones

Apartado b)

especular: a_bin → a_bin

func especular (a:a_bin) **dev** aie:a_bin
    si vacio?(a) **entonces** aie← Δ
    sino aie←especular(der(a))•raiz(a)•especular(izq(a))
    finsi
finfunc

Apartado d)

mayor_igual: nat a_bin → bool

func mayor_igual(n:natural,a:a_bin) **dev** b:bool
    si vacio?(a) **entonces** b←T
    sino
        b← **(**n>=(raiz(a)))
            ∧mayor_igual(n, izq(a))
            ∧ mayor_igual(n, der(a))
    finsi
finfunc

menor_igual: nat a_bin → bool

func menor_igual(n:natural,a:a_bin) **dev** b:bool
    si vacio?(a) **entonces** b←T
    sino
        b← **(**n<(raiz(a)))
            ∧menor_igual(n, izq(a))
            ∧ menor_igual(n, der(a))
    finsi
finfunc

esta_inorden?: a_bin → bool
func esta_inorden? (a:a_bin) **dev** b:bool
    si vacio?(a) **entonces** b←T
    sino b← esta_inorden?(izq(a))
        ∧ (mayor_igual(raíz(a), izq(a)))
        ∧ esta_inorden?(der(a))
        ∧ (menor_igual(raíz(a), der(a)))
    finsi
finfunc

Ejercicio 3.- Se quiere hacer un recorrido de un árbol por niveles (el nivel *k* son todos los nodos que están a distancia *k* de la raíz del árbol). Se pide:

a) nivel_n: a_bin natural → lista, que crea una lista con todos los nodos que se encuentren en el nivel indicado por el *natural* del segundo parámetro;

b) niveles_entre: a_bin natural natural → lista, que crea una lista con todos los nodos que se encuentren entre los niveles indicados por los dos números naturales; y

c) recorrer_niveles: a_bin → lista, que crea una lista formada por todos los niveles del árbol binario.

<u>Solución Operaciones</u>

Apartado a)

nivel_n: árbol→lista

func nivel-n(a:a_bin, n:natural) **dev** l:lista
    si vacio?(a) **entonces** l←[]
    sino **si** n=0 **entonces** l←[raíz(a)]
        sino l←nivel-n (izquierdo(a),n-1)
            ++ nivel-n(derecho(a),n-1)
    finsi
finfunc

Apartado b)

Niveles_entre: árbol→lista

func niveles-entre (a:árbol, n, m:natural)

dev l:lista

si (n<0) V (m<0) V (n>m)

entonces Error (`recorrido incorrecto')

sino si (n=m) **entonces** l←nivel_n(a,n)

sino l←niveles-entre(a, n, m-1)

++nivel-n(a,m)

finsi

finfunc

Apartado c)

recorrer_niveles: árbol→lista

func recorrer_niveles (a:árbol)**dev** l:lista

l←niveles-entre(a, 0, altura(a))

finfunc

Ejercicio 4.- Extender la especificación de los árboles generales vista en clase con las siguientes operaciones:

- num_nodos: árbol → natural, para calcular cuántos nodos hay en un árbol general;
- num_hojas: árbol → natural, para ver la cantidad total de hojas que tiene un árbol general;
- max_hijos: árbol → natural, que obtiene cuál es la mayor cantidad de hijos en un mismo nodo que hay en un árbol general.
- reflejar: árbol → árbol, que obtiene la imagen especular de un árbol;

- frontera: árbol → lista, que genera una lista formada por los elementos almacenados en las hojas del árbol, tomados de izquierda a derecha.

Solución Operaciones :

Apartado a)

num_nodos: árbol→natural

func num_nodos (a:árbol) **dev** n:natural

n← 1+ num_nodos_b (hijos(a))

finfunc

num_nodos_b: bosque→natural

func num_nodos_b (b:bosque) **dev** n:natural

var prim:arbol

si vacio?(b) **entonces** n←0

sino prim←primero(b)

n← num_nodos (prim)

\+ num_nodos _b(resto(b))

fin_si

finfunc

Apartado b)

num_hojas: árbol→natural

func num_hojas (a:árbol) **dev** n:natural

si vacio?(bosque(a)) entonces n← 1

sino n←num_hojas_b(hijos(a))

finfunc

num_hojas _b: bosque→natural
func num_hojas_b (b:bosque) **dev** n:natural

var prim:arbol

si vacio?(b) **entonces** n←0

sino

prim←primero(b)

n← num_hojas(prim)+num_hojas_b(resto(b))

fin_si

finfunc

Apartado c)
máx_hijos: árbol→natural
func máx_hijos (a:árbol) **dev** n:natural

var num_hijos, max_hijos_b:natural

num_hijos ← num_hijos(a)

max_hijos_b← max_hijos_b(bosque(a))

si (num_hijos > max_hijos_b)

entonces n← num_hijos

sino n← máx_hijos _b

finsi

finfunc

máx_hijos _b: bosque→natural

```
func  máx_hijos _b (b:bosque) dev n:natural
var   prim:arbol num_hijos_p, max_hijos_r:natural
          si vacio?(b) entonces max_hijos_b←0
                sino prim←primero(b)
                num_hijos_p← num_hijos(prim)
                max_hijos_r← max_hijos_b(resto(b))
                finsi
                si (num_hijos_p > max_hijos_r
                        entonces n← num_hijos_p
                sino n← max_hijos_r
                finsi
finfunc
```

Apartado d)

reflejar: árbol→árbol

```
func reflejar (a:árbol) dev ar:árbol
        ar←raiz(a)•reflejar_b(bosque(a))
finfunc
```

bosque_imagen:bosque→bosque

proc bosque_imagen (b, bimag:bosque)

{*procedimiento auxiliar que va creando el bosque imagen de b en bimag*}

var prim:árbol

mientras !vacio(b) **hacer** prim←primero(b)

b←resto(b)

bimag←reflejar(prim):bimag

finmientras

finsi

reflejar _b:bosque→bosque

func reflejar_b(b:bosque) **dev** br:bosque

var bimagen:bosque

bimagen←[]

bosque_imagen(b, bimagen)

{*procedimiento auxiliar que va creando el bosque imagen de b*}

br←bimag

finfunc

Apartado d)

frontera: árbol→lista

func frontera(a:árbol) **dev** l:lista

si num_hijos(a) =0 **entonces** l←[raíz(a)]

sino l←frontera_b(hijos(a))

finfunc

func frontera_b (b:bosque) **dev** l:lista

    si vacio?(b) **entonces** l←[]

    sino l←frontera(primero(b))++frontera_b(resto(b)

    finsi

finfunc

Ejercicio 5.- Llamaremos a un árbol general de naturales "maestro" si el valor de cada nodo es igual al número de hijos que tiene dicho nodo. Se pide:

a) Especificar completamente el TAD árbol general,
b) Comprobar si un árbol general es "maestro",
c) Buscar el nodo con mayor valor de un árbol maestro (es decir, el que tenga más hijos).

Ejercicio 6.-Llamaremos a un árbol general de naturales "creciente" en cada nivel del árbol, la cantidad de nodos que hay en ese nivel es igual al valor del nivel más uno; es decir, el nivel 0 tiene exactamente un nodo, el nivel 1 tiene exactamente dos nodos, el nivel *k* tiene exactamente *k* + 1 nodos. Se pide:

- Especificar completamente el TAD árbol general,
- Comprobar si un árbol general es "creciente",
- Buscar el nodo con mayor cantidad de hijos de un árbol creciente.

Necesitaremos una función auxiliar que cuente el total de nodos de un nivel dado k:

func nodos_nivel_k (a:árbol, k:natural) **dev** n:natural

    si k=0 **entonces** n←1

    sino n←nodos_nivel_k_b(hijos(a), k-1)

    finsi

finfunc

func nodos_nivel_k_b(b:bosque, k:natural) **dev** n:natural

    si vacio?(b) **entonces** n←0

sino si k=0 **entonces** n←long(b) {tamaño del bosque}

sino

n←nodos_nivel_k(primero(b), k) + nodos_nivel_k_b(resto(b), k)

finsi

finfunc

func creciente **(**a:árbol) **dev** b:boolean

creciente_desde_k(a,1)

finfunc

func creciente_desde_k (a: árbol,k:natural) **dev** b:boolean

si nodos_nivel_k(a, k) =0 **entonces** b←T

sino si nodos_nivel_k(a, k) !=k+1 **entonces** b←F

sino b←creciente_desde_k(a, k+1)

finsi

finfunc

o tb. {versión iterativa}

func creciente **(**a:árbol) **dev** b:boolean

k←0 es_creciente←T

mientras nodos_nivel_k(a, k) !=0 ∧ es_creciente **hacer**

si nodos_nivel_k(a, k) !=k+1 entonces es_creciente←F

sino k←k+1

finsi

finmientras

finfunc