FUNDAMENTOS DE COMPUTACIÓN
Repartido: Árboles

1. Árboles

Cuando hace un tiempo nos propusimos definir tipos con una cantidad no acotada de objetos, nos dimos cuenta de que era necesario disponer de un dispositivo para producir nuevos valores de esos tipos.

Así, definimos tipos infinitos a partir de **semillas** (que son los valores iniciales de estos tipos) y **generadores** (que producen valores **nuevos** a partir de objetos existentes) y con eso disparamos un proceso de generación de valores no acotados.

En esta última parte del curso nos dedicaremos a definir distintos tipos, que son las estructuras discretas donde se guarda la información necesaria para resolver problemas.

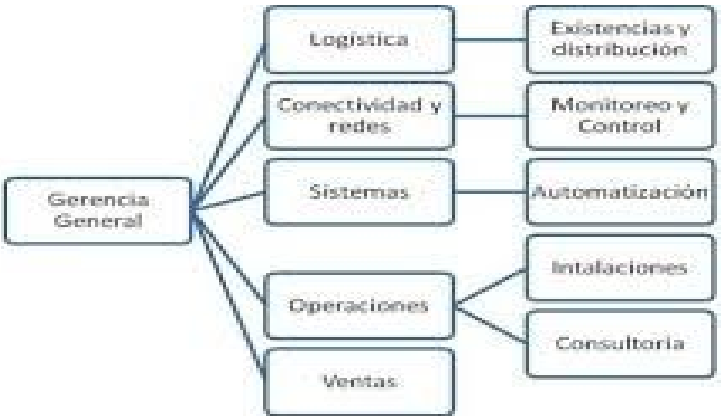
Los valores de los tipos que definiremos (y también de los que hemos definido!) pueden representarse gráficamente como estructuras llamadas **árboles**.

Un árbol es una estructura de datos generalmente no lineal compuesta de nodos, unidos por arcos. Los mismos son acíclicos y se utilizan para representar estructuras jerárquicas.

Hay muchos ejemplos de estas estructuras, como por ejemplo un organigrama de una empresa:

Cada elemento donde hay información en el árbol se llama **nodo o vértice**. El nodo superior del árbol se llama la **raíz**, y las líneas que unen dos nodos se llaman **arcos o aristas**. Cuando dos nodos se encuentran conectados por un arco, el nodo superior se llama **padre** y el inferior se llama **hijo.** Los nodos que no tienen hijos son las **hojas** del árbol. Los nodos que no son hojas se llaman **nodos internos.**

Los árboles pueden dibujarse de cualquier forma, pero lo usual es hacerlo con la raíz arriba, sus hijos en el siguiente nivel, y así sucesivamente hasta llegar a las hojas, que se encuentran debajo del todo. Pero también podríamos dibujar al árbol en cualquier otro sentido.



Los árboles son un caso particular de estructuras llamadas grafos, las cuales se utilizan en matemática para representar relaciones binarias. La característica que define un árbol es que **todos los nodos menos la raíz tienen exactamente un padre**.

Hay distintos tipos de árboles, con información en los nodos que puede ser de distinta clase. Se puede guardar información de cierto tipo en los nodos internos y de otro tipo en las hojas, o no poner información en algunos nodos, etc.

2. Árboles Binarios

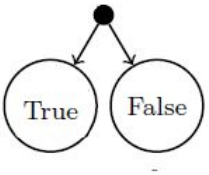
Los árboles binarios se caracterizan porque cada nodo interno tiene exactamente dos hijos. Definimos el siguiente tipo en Haskell:

data AB a where { Hoja :: a → AB a ;
Nodo :: AB a → AB a → AB a }

Se trata de un tipo que tiene un parámetro **a** de tipo, al igual que las listas.

Los elementos del tipo **AB a** se construyen a partir de la semilla **Hoja**, que construye un nodo hoja con información adentro de tipo **a.**
Así, por ejemplo, podemos construir los objetos **Hoja False** y **Hoja True** de tipo **AB Bool**, que podemos representar gráficamente del siguiente modo:

El generador **Nodo** toma dos árboles y construye uno nuevo, con una nueva raíz (pero sin agregar información en el nodo interno).
Así, podemos unir dos árboles **Hoja False** y **Hoja True** y obtener un nuevo árbol:

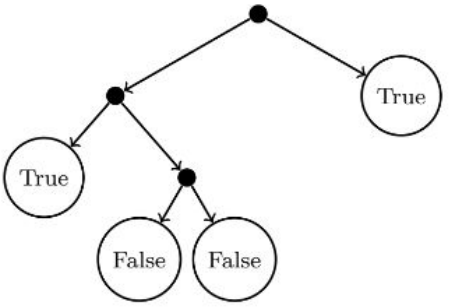


que corresponde a la expresión **Nodo (Hoja True) (Hoja False) :: AB Bool.**

Así, podemos seguir aplicando el constructor **Nodo** a árboles ya definidos, y obtener nuevos árboles.

Por ejemplo, el siguiente árbol se corresponderá con la expresión:

Nodo (Nodo (Hoja True) (Nodo (Hoja False) (Hoja False))) (Hoja True) :: AB Bool



Expresión case

Asociado al tipo definido tenemos la expresión **case** correspondiente, cuya regla será:

$$\frac{e :: AB\ a \qquad e1 :: t \ \ [x :: a] \qquad e2 :: t\ [\ t1 :: AB\ a\ ,\ t2 : AB\ a\ ]}{case\ e\ of\ \{\ Hoja\ x \rightarrow e1\ ;\ Nodo\ t1\ t2 \rightarrow e2\ \} :: t}$$

y las igualdades correspondientes:

case (Hoja e) of { Hoja x → e1 ; Nodo t1 t2 → e2 } = e1 [x:=e]
case (Nodo ti td) of { Hoja x → e1 ; Nodo t1 t2 → e2 } = e2 [t1 := ti , t2 := td]

Haciendo uso de la expresión **case** podemos definir algunas funciones sobre árboles binarios:

cantHojas :: AB a → N, que calcula la cantidad de hojas que tiene un árbol binario.

```
cantHojas =  λt→ case t of {
                        Hoja x → S O;
                        Nodo ti td → cantHojas ti + cantHojas td  }
```

cantNodos :: AB a → N, que calcula la cantidad de nodos internos que tiene un árbol binario:

```
cantNodos =  λt→ case t of {
                        Hoja x → O;
                        Nodo ti td → S (cantNodos ti + cantNodos td)  }
```

Otras posibles funciones son:

1. **hojas :: AB a → [a]**, que devuelve una lista con las hojas de un árbol binario**.**
2. **espejo :: AB a → AB a**, que devuelve el´arbol espejo de un´arbol binario (o sea, otro con los mismos elementos, pero con todos los sub árboles transpuestos).
3. **maxHoja :: Ord a => AB a → a**, que calcula el máximo de las hojas de un árbol binario.
4. **mapAB :: (a→b) → AB a → AB b**, que le aplica una función a todas las hojas de un árbol binario.
5. **altura :: AB a → N**, que calcula la altura del árbol, esto es, la longitud del camino más largo desde la raíz a una hoja.

Inducción Primitiva

Asimismo, para demostrar afirmaciones del tipo **(∀ t :: AB a) P(t)** justificamos un principio de inducción para el tipo de los árboles binarios.

Para demostrar (∀ t :: AB a) P(t) es suficiente demostrar:

1. **<u>Caso Hoja</u>: (∀x :: a) P(Hoja x) se cumple.**

2. **<u>Caso Nodo</u>: (∀ ti, td :: AB a) P(ti) y P(td) ⇒ P(Nodo ti td) se cumple.**

De este modo, las demostraciones sobre árboles binarios serán de la forma:

(∀t :: AB a) P(t).
Dem. Por inducción en t :: AB a
Caso Hoja: (∀x :: a) P (Hoja x)
Dem. ....
.....
.....

Caso Nodo: Sean ti, td :: AB a
HI1) P (ti)
HI2) P(td)
TI) P(Nodo ti td)
Dem. ....
.....
.....

Una propiedad de estos árboles binarios que podemos demostrar como ejemplo es la siguiente:

(∀t :: AB a) cantHojas t = S(cantNodos t).

Observar que la propiedad a demostrar es P = cantHojas ● = S(cantNodos ●)

Dem. Por inducción en t :: AB a
Caso Hoja: (∀x :: a) cantHojas (Hoja x) =? S(cantNodos (Hoja x))
Dem. Sea x :: a.

cantHojas (Hoja x)	| S (cantNodos (Hoja x))
= (def. cantHojas, β, case)	| = (def. cantNodos, β, case)
S O	| S O

Ambas expresiones son iguales por reducción a la misma expresión.

Caso Nodo: Sean t1, t2 :: AB a
HI1) cantHojas t1 = S(cantNodos t1)
HI2) cantHojas t2 = S(cantNodos t2)
TI) cantHojas (Nodo t1 t2) =? S (cantNodos (Nodo t1 t2))
Dem.

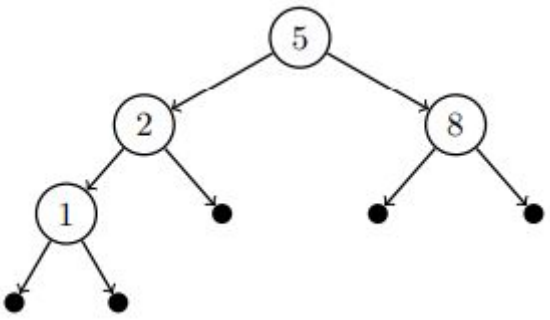
cantHojas (Nodo t1 t2)	| S (cantNodos (Nodo t1 t2))
= (def. cantHojas, β, case)	| = (def. cantNodos, β, case)
cantHojas t1 + cantHojas t2	| S (S (cantNodos t1 + cantNodos t2))
= (HI1 & HI2)	|
S(cantNodos t1) + S(cantNodos t2)	
= (def. (+), $Lema_{+S}$)	
S (S (cantNodos t1 + cantNodos t2))	

Ambas expresiones son iguales por reducción a la misma expresión.

3. Árboles Binarios de Búsqueda

Los árboles binarios de búsqueda se utilizan como estructuras donde guardar datos en cierto orden que explicaremos a continuación, y donde la búsqueda, inserción y clasificación son muy eficientes.

Comenzamos definiendo el tipo **ABB a** de árboles binarios, donde cada nodo interno tiene exactamente dos hijos e información de tipo **a**. Las hojas no tienen información.



Definimos el tipo como sigue:

data ABB a where { Vacio :: ABB a ; Unir :: ABB a → a → ABB a → ABB a }

Así, el árbol de arriba se corresponderá con la siguiente expresión:

Unir(Unir (Unir Vacio 1 Vacio) 2 Vacio) 5 (Unir Vacio 8 Vacio) :: ABB a

Podemos definir funciones que devuelvan una lista con los nodos internos del árbol, recorriendolo de izquierda a derecha, pero de distintas formas, según se listen las raíces de los subárboles:

In-order: lista los elementos de un ABB de izquierda a derecha, de modo tal que para cada nodo, aparecen primero en la lista todos los elementos que están a su izquierda, luego el nodo y después los que están a su derecha (siguiendo el mismo criterio).

Pre-order: difiere del anterior en que para cada nodo, primero aparece él mismo, luego la lista de todos los elementos que están a su izquierda y después la de los que están a su derecha (siguiendo el mismo criterio).

Post-order: para cada nodo se lista primero los elementos que están a su izquierda, luego los elementos que están a su derecha (siguiendo el mismo criterio) y finalmente el mismo nodo.

Definimos entonces las siguientes funciones:

```
inOrder :: ABB a → [a]
inOrder = λt→ case t of { Vacio → [ ] ;
                          Unir ti x td → inOrder ti ++ x : inOrder td }
```

dejamos la definición de las otras funciones como ejercicio:

```
preOrder :: ABB a → [a]
postOrder :: ABB a → [a]
```

Los árboles binarios de búsqueda se utilizan para guardar y ordenar información, y para ello se define cuándo un ABB está **ordenado**.
Esto es, cuando para cada uno de sus nodos se cumple la siguiente propiedad: **todos los nodos a su izquierda son menores o iguales a él, y todos los que están a su derecha son mayores que él.**
Observar que el árbol del ejemplo está ordenado.

Definimos la función que decide si un ABB está ordenado:

```
ordenado :: Ord a => ABB a → Bool
ordenado = λt→ case t of { Vacio → True ;
                           Unir ti x td → menIg ti x && mayores td x
                                          && ordenado ti && ordenado td }
```

Las funciones **menIg** y **mayores** verifican que todos los nodos de un ABB sean menores o iguales y mayores respectivamente que un elemento (en este caso las utilizaremos con la raíz del árbol).

En vez de definir dos funciones, podemos definir una sola, que reciba como parámetro la relación que debe complirse entre los nodos del ABB y el elemento dado.
Entonces definiremos:

```
cumple :: (a → a → Bool) → ABB a → a → Bool
cumple = λr t x. case t of { Vacio → True ;
                             Unir ti z td → r x z && cumple r ti x && cumple r td x }
```

Con esta función, podemos definir:

```
menig :: Ord a => ABB a → a → Bool
menig = cumple (<=)

mayores :: Ord a => ABB a → a → Bool
mayores = cumple (>)
```

Tree Sort

Los árboles binarios de búsqueda se utilizan para ordenar listas, definiendo un método de ordenamientos llamado Tree Sort.
La propiedad esencial que se utiliza es que si el ABB está ordenado, entonces el resultado de recorrerlo en in-order es una lista ordenada.

El método de ordenamiento con ABB consiste en ir insertando recursivamente uno por uno los elementos de una lista en un ABB, de modo tal de obtener un árbol ordenado.
Luego, se listan los elementos del ABB utilizando la función inOrder y se obtiene una lista ordenada.

Ejercicios
Defina las siguientes funciones para implementar el algoritmo de Tree Sort:

- **insertABB :: Ord a => a → ABB a → ABB a**, que inserta un elemento en un ABB ordenado, de modo tal que el árbol resultante también queda ordenado.
- **list2ABB :: Ord a => [a] → ABB a**, que genera una ABB ordenado a partir de una lista, insertando los elementos uno por uno.
- **treeSort :: Ord a => [a] → [a]**, que ordena una lista insertando sus elementos en un ABB ordenado y luego listando sus nodos con inOrder.