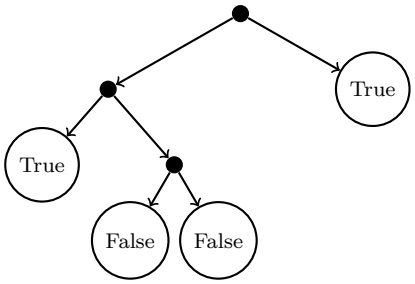
FUNDAMENTOS DE COMPUTACIÓN
PRÁCTICO 6
ÁRBOLES

Ejercicio 1. Considere la siguiente definición del tipo (polimórfico) `AB a` de árboles binarios con información de tipo `a` en las hojas:

```
data AB a where { Hoja ::  a -> AB a ;
                  Nodo ::  AB a -> AB a -> AB a}
```

(a) Codifique el siguiente árbol como una expresión `to` de Haskell y dé su tipo:



Defina las siguientes funciones utilizando recursión sobre el tipo `AB`:

(b) `cantHojas :: AB a -> N`, que calcula la cantidad de hojas que tiene un árbol binario.
(c) `cantNodos :: AB a -> N`, que calcula la cantidad de nodos internos que tiene un árbol binario.
(d) `hojas :: AB a -> [a]`, que devuelve una lista con las hojas de un árbol binario.
(e) `altura :: AB a -> N`, que calcula la altura de un árbol binario, es decir la longitud de alguna de sus ramas más largas (puede haber más de una).
(f) `espejo :: AB a -> AB a`, que devuelve el árbol espejo de un árbol binario (o sea, otro con los mismos elementos, pero con todos los subárboles transpuestos).
(g) `maxHoja :: Ord a => AB a -> a`, que calcula el máximo de las hojas de un árbol binario.
(h) `mapAB :: (a->b) -> AB a -> AB b`, que le aplica una función a todas las hojas de un árbol binario.

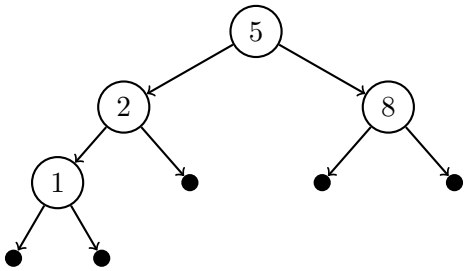
Ejercicio 2. Demuestre las siguientes propiedades por inducción en `t :: AB a`:

(a) (∀`t :: AB a`) `cantHojas t = S(cantNodos t)`.
(b) (∀`t :: AB a`)(∀`f :: a->b`) `cantNodos (mapAB f t) = cantNodos t`.
(c) (∀`t :: AB a`) `espejo (espejo t) = t`.
(d) (∀`t :: AB a`) `hojas (espejo t) = reverse (hojas t)` .

Ejercicio 3. Considere el tipo (polimórfico) `ABB a` de árboles binarios con información de tipo `a` en los nodos internos:

```
data ABB a where { Vacio ::  ABB a ;
                   Unir ::  ABB a -> a -> ABB a -> ABB a}
```

Codifique el siguiente árbol como una expresión `to` de Haskell y dé su tipo:



Defina las siguientes funciones sobre `ABB`s:

(a) `inOrder :: ABB a -> [a]`, la cual, dado un `ABB` lista sus elementos de izquierda a derecha: 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).

(b) `preOrder :: ABB a -> [a]`, que lista los nodos internos de un `ABB` del siguiente modo: 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).

(c) `postOrder :: ABB a -> [a]`, que lista los nodos internos de un `ABB` del siguiente modo: para cada nodo, primero la lista de los elementos que están a su izquiera, luego los elementos que están a su derecha (siguiendo el mismo criterio) y finalmente el mismo nodo.

(d) `cantNodosABB :: ABB a -> N`, que calcula la cantidad de nodos internos que tiene un `ABB`.

(e) `pertenece :: Eq a => a -> ABB a -> Bool` que verifica si un elemento pertence a un `ABB`.

(f) `espejoABB :: ABB a -> ABB a`, que devuelve el árbol espejo de un `ABB`.

Ejercicio 4. Los árboles binarios de búsqueda se utilizan para ordenar listas, definiendo un método de ordenamientos llamado *Tree Sort*. 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.* La propiedad que se cumple es que si el `ABB` está ordenado, entonces el resultado de recorrerlo en inorden 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 recorre y se listan los elementos del `ABB` utilizando la función `inOrder` definida en el ejercicio 3.

Defina las siguientes funciones sobre árboles binarios de búsqueda:

(a) `ordenado :: Ord a => ABB a -> Bool`, que verifica si un `ABB` está ordenado (ver definición arriba).

(b) `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.

(c) `list2ABB :: Ord a => [a] -> ABB` a que genera una `ABB` ordenado a partir de una lista, insertando los elementos uno por uno.

(d) `treeSort :: Ord a => [a] -> [a]`, que ordena una lista insertando sus elementos en un `ABB` ordenado y luego listando sus nodos con `inOrder`.

Ejercicio 5. Demuestre por inducción las siguientes propiedades sobre árboles de tipo ABB:

(a) `(∀t :: ABB a) postOrder (espejoABB t) = reverse (preOrder t)`
(b) `(∀t :: ABB a) length(preOrder t) = cantNodos t`

Ejercicio 6. Para representar expresiones aritméticas simples se utiliza el siguiente tipo de árboles, donde las hojas son números enteros y los nodos internos se corresponden con las operaciones de suma y multiplicación:

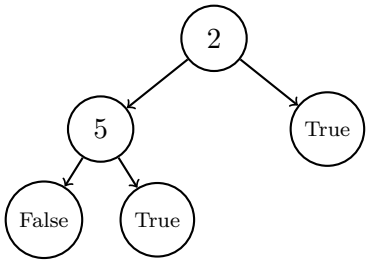
```
data Exp where { Num  ::  Int -> Exp;
                 Sum  ::  Exp -> Exp -> Exp;
                 Mul  ::  Exp -> Exp -> Exp }
```

(a) Codifique la expresión $(2 + 3) * (4 * (5 + 1))$ como un elemento de tipo `Exp`.
(b) Defina una función `eval :: Exp -> Int`, que calcula el valor de una expresión. Se podrán usar las operaciones correspondientes sobre enteros.
Por ejemplo, para la expresión del ejercicio anterior, el resultado de aplicar `eval` deberá ser 120.
(c) Defina una función `set :: Exp -> Int -> Exp`, que reemplaza todas las hojas de una expresión por un número entero dado, dejando el resto de la expresión sin cambiar.
(d) Demuestre que: `(∀e :: Exp) eval(set e 0) = 0`

Ejercicio 7. Considere el siguiente tipo `Tab` de árboles con nodos internos de tipo `a` y hojas de tipo `b`:

```
data Tab where { H  ::  b -> Tab;
                 N  ::  Tab -> a -> Tab -> Tab}
```

(a) Codifique el siguiente árbol como una expresión `e`, y dar su tipo:

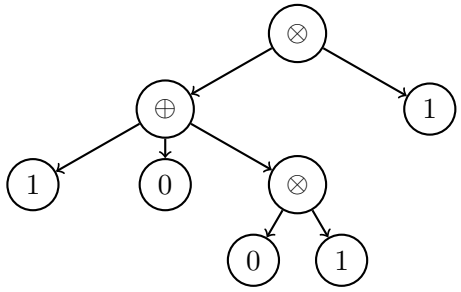


(b) Defina la función `splitT :: Tab -> ([a],[b])` que recibe un árbol `t` y devuelve un par de listas conteniendo los nodos internos y las hojas de t respectivamente.

Ejercicio 8. Considere el siguiente tipo de árboles, donde las hojas son bits (`0` y `1`), y los nodos internos son de dos tipos: $\otimes$ y $\oplus$, y pueden tener una cantidad arbitraria de hijos (que se agrupan en una lista):

```
data BT where { 0  :: BT ;
                1  :: BT ;
                ⊗  :: [BT] -> BT ;
                ⊕  :: [BT] -> BT }
```

(a) Codifique el siguiente árbol como una expresión de tipo `BT`:



(b) Defina una función `unos :: BT -> Int` que calcule cuantos `1`s tiene un árbol de bits. En el ejemplo el resultado sería 3. Puede utilizar la suma de enteros `(+)`.

(c) Defina una función `valor :: BT -> Bool` que calcule el valor de un árbol de bits, teniendo en cuenta que:

- Los nodos `0` tienen valor `False`
- Los nodos `1` tienen valor `True`
- Los nodos $\otimes$ tienen valor `True` sólo si *todos* sus hijos tienen valor `True`
- Los nodos $\oplus$ tienen valor `True` si *alguno* de sus hijos tiene valor `True`

Para el árbol del ejemplo, el resultado de la función `valor` debe dar `True`.
Puede utilizar las funciones `(&&),(||) :: Bool->Bool->Bool` y definir las funciones auxiliares que considere necesarias.

Ejercicio 9. Considere la siguiente definición del tipo `LArbol`, de árboles n-arios, esto es, cada nodo interno puede tener una cantidad arbitraria de hijos, que se agrupan en una lista:

```
data LArbol a where { Nodo ::  a -> [LArbol a] -> LArbol a}
```

Defina las siguientes funciones para elementos del tipo `LArbol`.
Recomendamos fuertemente utilizar las funciones ya definidas para listas.

(a) `cantNodosLA :: LArbol a -> Int`, que calcula la cantidad de nodos de un `LArbol`.

(b) `alturaLA :: LArbol a -> Int`, que calcula la altura de un `LArbol`.

(c) `mapLA :: (a->b) -> LArbol a -> LArbol b`, que le aplica una función a todos los nodos de un `LArbol`.

(d) `aridad :: LArbol a -> Int`, que calcula la aridad de un `LArbol`, esto es, el número máximo de hijos que tienen sus nodos.

(e) `larbol2list :: LArbol a -> [a]` que devuelve una lista con todos los elementos de un LArbol.

Ejercicio 10. Considere la siguiente definición:

```
g :: [a] -> T a -> Bool
g = \m -> \t -> case t of {
                      P y z -> case m of { [] -> y ; x:xs -> g xs z};
                      Q q s -> g s q;
                      R -> False}
```

(a) Defina el tipo `T a` para que la función `g` compile: `data T a where {P::...;Q::...;R::...}`

(b) Defina la función `listar :: T a -> [a]`, que devuelve una lista conteniendo todos los elementos de las listas que haya en los nodos de un árbol.
Puede utilizar la concatenación de listas `(++) :: [a] -> [a] -> [a]`, definida como
`(++) = \l1 l2 -> case l1 of {[] -> l2 ; x:xs -> x : (xs ++ l2)}`.

(c) Defina la función `andT :: T Bool -> Bool`, que calcula la conjunción (el `&&`) de todos los booleanos de los nodos de un árbol de tipo `T Bool`, *incluyendo* los que están en las listas.
Puede utilizar la función `and :: [Bool] -> Bool` que calcula la conjunción de los elementos de una lista de booleanos, definida como: `and = \l -> case l of {[] -> True; b:bs -> b && and bs}`.

Ejercicio 11. Considere la siguiente definición:

```
g :: Arb a b -> a -> b
g = \t e -> case t of { M x z -> g z (x e);
                        P u v -> case u of {True -> snd v ; False -> fst v};
                        Q h i j k -> case h of {O -> g i e ; S x -> g j k} }
```

donde `fst :: (a,b) -> a`, `snd :: (a,b) -> b` se definen como:

```
fst = \p -> case p of {(x,y) -> x}
snd = \p -> case p of {(x,y) -> y}
```

(a) Defina el tipo `Arb a b` para que la función `f` compile: `data Arb a b where {M::...;P::...;Q::...}`

(b) Defina la función `cantb :: Arb a b -> N` que recibe un árbol de tipo `Arb a b` y calcula la cantidad de elementos de tipo `b` que hay en el árbol.
Puede utilizar la suma de naturales `(+) :: N -> N -> N`.

(c) Defina la función `cantP :: Arb a b -> N` que dado un árbol de tipo `Arb a b`, calcula la cantidad de nodos `P` que hay en él.

(d) Demuestre `(∀t :: Arb a b) cantb t = doble (cantP t)`, donde `doble::N -> N` se define como `doble = \n -> case n of {O -> O ; S x -> S (S (doble x))}`.
Puede utilizar el siguiente resultado sin necesidad de demostrarlo:
L_{doble}: `(∀m,n::N) doble (m + n) = doble m + doble n`.

Ejercicio 12. Se define el siguiente tipo de listas con elementos de tipo `a` y `b` indistintamente:

```
data L a b where { V  :: L a b;
                   NA :: a -> L a b -> L a b;
                   NB :: b -> L a b -> L a b }
```

(a) Escriba una expresión `e` de tipo `L` que contenga a los siguientes elementos: `2,True,3,6` (en ese orden), y dé su tipo.

(b) Defina la función `cantNA :: L a b -> N`, que recibe una lista `l` de tipo `L a b` y devuelve la cantidad de elementos de tipo `a` que contiene.
Por ejemplo, para la lista del punto anterior, se cumple: `cantNA e = 3`.

(c) Defina la función `filtrar :: L a b -> (a->Bool) -> (b->Bool) -> L a b` , que recibe una lista `l` y dos predicados `p` y `q` sobre elementos de tipo `a` y `b` respectivamente, y devuelve la lista con los elementos de `l` que cumplen el predicado correspondiente a su tipo.
Por ejemplo, `filtrar e par id` contendrá los elementos 2, `True` y 6 solamente.

(d) Demuestre que $(\forall l :: L\,a\,b, \forall p :: a \to Bool, \forall q :: b \to Bool)\ cantNA\ (filtrar\ l\ p\ q) \leq cantNA\ l$

Puede utilizar los siguientes resultados sin necesidad de demostrarlos:

L1. $(\forall x :: N)\ x \leq x$
L2. $(\forall x :: N)\ x \leq S\ x$
L3. $(\forall x, \forall y :: N)\ x \leq y \Leftrightarrow S\ x \leq S\ y$
L4. Transitividad de $\leq$.

Ejercicio 13. Considere el siguiente tipo `P a`, de listas con cantidad par de elementos de tipo `a`:

```
data P a where { Nada   :: P a ;
                 DosMas :: a -> a -> P a -> P a }
```

(a) Escriba una expresión `l` de tipo `P Int` que represente a una lista que contiene a los números del `0` al `3` (en ese orden).

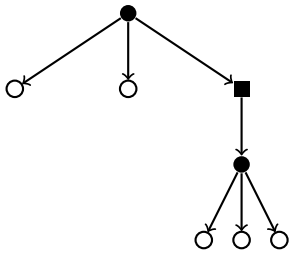
(b) Programar la función `swap :: P a -> P a` que recibe una lista de tipo `P a` y devuelve otra donde cada pareja de elementos adyacentes está intercambiado. Por ejemplo, si la lista contiene los elementos `0,1,2` y `3`, la función `swap` deberá devolver otra con los números `1,0,3` y `2`.

(c) Demostrar que `(∀l :: P a) (swap (swap l) = l)`

Ejercicio 14. Considere el siguiente tipo `A` de árboles con uno o tres hijos en cada nodo interno:

```
data A where { H  ::  A;
               U  ::  A -> A;
               T  ::  A -> A -> A -> A }
```

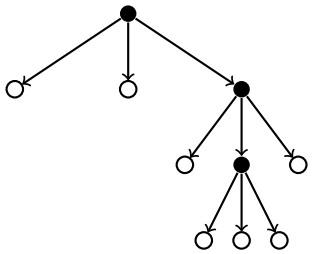
(a) Escriba la expresión Haskell correspondiente al siguiente árbol:



(b) Defina la función `nodos :: A -> N`, que calcula la cantidad total de nodos (internos y hojas) de un árbol de tipo `A`.
Para el árbol del ejemplo, la cantidad de nodos es 8.

(c) Defina la función `unos :: A -> N`, que calcula la cantidad de nodos de un árbol de tipo `A` que tienen un solo hijo.
En el árbol del ejemplo, hay solamente un nodo que cumple esa propiedad (el nodo dibujado con un cuadrado).

(d) Defina la función `trans :: A -> A`, que reemplaza los nodos que tiene un hijo por un nodo con tres hijos, agregando una hoja en cada extremo nuevo.
Para el árbol del ejemplo, el resultado será:



(e) Demuestre que `(∀a :: A) nodos a` $\leq$ `nodos (trans a)` ,

(f) Demuestre que `(∀a :: A) nodos (trans a) = nodos a + doble (unos a)`,
donde `doble :: N -> N` se define como:
`doble = \x -> case x of { O -> O; S x -> S (S (doble x))}`

Puede utilizar la suma de naturales `(+)` y los lemas de `(+)` y $\leq$ que considere necesarios, sin necesidad de demostrarlos.