Práctiques de Llenguatges, Tecnologies i Paradigmes de programació. Curs 2015-16

Part II Programació Funcional

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Pràctica 4: Tipus algebraics i ordre superior

Índex

1 Objectiu de la Pràctica **2**

2 Tipus i Inferència de Tipus **2**

3 Estratègia de Reducció **3**

4 Les llistes **4**
- 4.1 El tipus `lista` 4
- 4.2 Els rangs 8
- 4.3 Les llistes intensionales 8

5 Les funcions `map` i `filter` **9**

6 Tipus algebraics **11**
- 6.1 Enumeracions 11
- 6.2 Tipus Renomenat 11
- 6.3 Arbres 12

7 Avaluació **15**

1 Objectiu de la Pràctica

En aquesta pràctica es presenta:

- El maneig de llistes i llistes intensionales en GHCi, així com les funcions `map` i `filter`.
- El maneig d'estructures de dades mitjançant tipus construïts (tipus algebraics) en Haskell.

S'han previst 3 sessions per a resoldre els exercicis plantejats en aquesta pràctica.

2 Tipus i Inferència de Tipus

En programació funcional, els valors que poden obtenir-se en avaluar expressions estan organitzats per *tipus*. Cada tipus representa un conjunt de valors. Per exemple, el tipus (primitiu o bàsic) `Int` representa els valors `0`, `1`, `2`, `-1`, `-2`, etc. El tipus `Bool` representa els valors booleans `True` i `False`. El tipus `Char` els caràcters alfanumèrics `'a'`, `'b'`, `'A'`, `'B'`, `'0'`, ... En programació funcional, com en la majoría de llenguatges de programació, *tota expressió té associat un tipus*. Açò es pot expressar explícitament mitjançant l'operador de tipificació '`::`'

```
42 :: Int
6*7 :: Int
doble (2+2) :: Int
```

Com ja s'ha vist, es pot demanar a l'intèrpret GHCi que informe sobre el tipus de qualsevol expressió ben formada usant el comando d'usuari "`:t`":

```
> :t 'a'
'a' :: Char


> :t "Hello"
"Hello" :: String
```

En programació funcional, els identificadors de les funcions són expressions vàlides gràcies a la currificació i, com a tals, tenen un tipus. A més, no és necessari que el programador proporcione informació explícita de tipificació de les funcions que definisca o de les expressions utilitzades, ja que l'intèrpret el *infereix* automàticament a partir de les definicions.

Es pot provar, per exemple:

```
> :t show
> :t (+)
> :t (*)
> :t (3 +)
```

```
> :t (* 2.0)
```

Nota addicional: Observe's el cas particular de la funció binària `(*)` que espera dos arguments d'un tipus numèric i retorna un valor del mateix tipus numèric.

```
Prelude> :t (*)
(*) :: (Num a) => a -> a -> a
```

No obstant açò, la funció unaria `(* 2.0)` espera només un argument, a més restringit al tipus `Float` o algun semblant, i retorna el seu doble, és a dir, el nombre que resulta de multiplicar l'argument per la constant real 2.0.

```
Prelude> :t (* 2.0)
(* 2.0) :: (Fractional a) => a -> a
```

Observe's que aquesta funció s'ha obtingut com una simple aplicació parcial de l'operador de multiplicació `(*)`, que és binari, al cas particular d'haver fixat el seu segon argument perquè prenga el valor de la constant real 2.0, la qual cosa és possible gràcies a que aquesta funció està definida de forma currificada.

3 Estratègia de Reducció

L'estratègia de reducció en Haskell és *lazy* (mandrosa). Aquesta estratègia redueix una expressió (parcialment) només si realment és necessari per a calcular el resultat. És a dir, es redueixen els arguments només prou per a poder aplicar algun pas de reducció en el símbol de funció més extern.

Gràcies a açò, és possible treballar amb estructures de dades infinites. En llenguatges que usen l'estratègia *eager* (impacient), com són la majoría de els llenguatges imperatius i els llenguatges funcionals més antics, aquest tipus d'estructures no poden manipular-se.

La funció `repeat'` retorna una llista infinita (similar a la funció `repeat` del `Prelude`):

```
repeat' :: a -> [a]
repeat' x = xs where xs = x:xs
```

L'avaluació del terme `repeat' 3` retorna la llista infinita `[3,3,3,3,3,3,3,3,3,...`

Nota addicional: S'ha d'anar amb compte perqu la crida `repeat' 3` no acaba i imprimeix una llista infinita del nmero 3 per la pantalla, havent

de parar la l'execució amb `^C`.

Una llista infinita generada per `repeat'` pot ser usada com a argument parcial per una funció que té un resultat finit. La segent funció, per exemple, pren un nombre finit d'elements d'una llista:

```
take' :: Int -> [a] -> [a]
take' _ [] = []
take' n (x:t)
   | n<=0 = []
   | otherwise = x : take' (n - 1) t
```

Per exemple, la trucada `take' 3 (repeat' 4)` retorna la llista `[4,4,4]`.

4 Les llistes

4.1 El tipus `lista`

En programació funcional és possible emprar tipus estructurats els valors dels quals estan compostos per objectes d'altres tipus. Per exemple, el tipus llista `[a]` pot servir per a aglutinar en una única estructura objectes del *mateix* tipus (denotat, en aquest cas, per la variable de tipus `a`, que pot instanciarse a qualsevol tipus). En Haskell, les llistes poden especificar-se tancant els seus elements entre claudàtors i separant-los amb comes:

```
[1,2,3] :: [Int]
['a','b','c','d'] :: [Char]
[cos,log,(1.0 +)] :: [Float -> Float]
         -- Llista de funcions de
         -- reals en reals: cosinus, logaritme en base 2,
         -- incrementar una unitat un nombre real.
```

No obstant açò, les llistes

```
[1,'a',2]
['a',log,3]
[cos,2,(*)]
```

no són vàlides (per què? què respon GHCi en preguntar pel tipus?).

La llista buida es denota com `[]`. Quan no és buida, es pot descompondre usant una notació que separa l'element inicial, de la llista que conté els elements restants. Per exemple:

```
1:[2,3]   o   1:2:[3]   o   1:2:3:[]
'a':['b','c','d']  o  'a':'b':['c','d'] o 'a':'b':'c':'d':[]
cos:[log]   o   cos:log:[]
```

Els tipus que inclouen variables de tipus en la seua definició (com el tipus llista) són tipus genèrics o *polimòrfics*. Es poden definir funcions sobre tipus polimòrfics, que poden emprar-se sobre objectes de qualsevol tipus que siga una instància dels tipus polimòrfics involucrats. Per exemple, la funció (predefinida) `length` calcula la longitud d'una llista:

```
> :t length
length :: [a] -> Int
```

La funció `length` pot emprar-se sobre llistes de qualsevol tipus base:

```
> length [1,2,3]
3
> length ['a','b','c','d']
4
> length [doble,cuadruple,fact]
3
```

L'operador `(!!)` permet la indexació de llistes. S'usa per a obtenir l'element en una posició donada d'una llista:

```
> :t (!!)
(!!) :: [a] -> Int -> a la
```

indexació pot utilitzar-se amb llistes de qualsevol tipus base:

```
> [1,2,3] !! 2
3
>  ['a','b','c','d'] !! 0
'a'
```

Una altra funció molt útil per a llistes és `(++)`, que s'usa per a concatenar dues llistes de qualsevol tipus:

```
> [1,2,3] ++ [4,5,6]
[1,2,3,4,5,6]
> ['a','b','c','d'] ++ ['e','f']
"abcdef"
```

Exercici 1 (Resolt) *Definir una funció per a calcular el valor binari corresponent a un nombre enter no negatiu x:*

```
decBin :: Int -> [Int]
```

Per exemple, l'avaluació de l'expressió:

```
> decBin 4
```

ha de retornar la llista `[0,0,1]` *(començant pel bit menys significatiu).*

```
module DecBin where
 decBin :: Int -> [Int]
 decBin x = if x < 2 then [x]
            else (x `mod` 2) : decBin (x `div` 2)
```

Exercici 2 (Resolt) *Definir una funció per a calcular el valor decimal corresponent a un nombre en binari (representat com una llista d'1's i 0's):*

```
binDec :: [Int] -> Int
```

Per exemple, l'avaluació de l'expressió:

```
> binDec [0,1,1]
```

ha de retornar el valor `6`.

```
module BinDec where
 binDec :: [Int] -> Int
 binDec (x:[]) = x
 binDec (x:y)  = x + binDec y * 2
```

Exercici 3 *Definir una funció per a calcular la llista de divisors d'un nombre enter no negatiu n:*

```
divisors :: Int -> [Int]
```

Per exemple, l'avaluació de l'expressió:

```
> divisors 24
```

ha de retornar la llista `[1,2,3,4,6,8,12,24]`.

Exercici 4 *Definir una funció per a determinar si un enter pertany a una llista d'enters:*

```
member :: Int -> [Int] -> Bool
```

Per exemple, l'avaluació de l'expressió:

```
> member 1 [1,2,3,4,8,9]
```

ha de retornar el valor `True`. *I l'avaluació de l'expressió:*

```
> member 0 [1,2,3,4,8,9]
```

ha de retornar el valor `False`.

Exercici 5 *Definir una funció per a comprovar si un nombre és primer (els seus divisors són 1 i el propi nombre) i una funció per a calcular la llista dels n primers nombres primers:*

```
isPrime :: Int -> Bool
primes :: Int -> [Int]
```

Per exemple, l'avaluació de l'expressió:

```
> isPrime 2
```

ha de retornar el valor `True`. *I l'avaluació de l'expressió:*

```
> primes 5
```

ha de retornar la llista `[1,2,3,5,7]`. *Es recorda que Haskell permet obtenir fàcilment una llista amb els elements inicials d'una altra llista infinita (com, per exemple, la llista de tots els nombres primers).*

Exercici 6 *Definir una funció per a determinar quantes vegades apareix repetit un element en una llista d'enters:*

```
repeated :: Int-> [Int] -> Int
```

Per exemple, l'avaluació de l'expressió:

```
> repeated 2 [1,2,3,2,4,2]
```

ha de retornar el valor `3`.

Exercici 7 *Definir una funció per a* concatenar *llistes de llistes, que prenga com a argument una llista de llistes i retorne la concatenació de les llistes considerades:*

```
concat' :: [[a]] -> [a]
```

Per exemple, l'avaluació de l'expressió:

```
> concat' [[1,2],[3,4],[8,9]]
```

ha de retornar la llista `[1,2,3,4,8,9]`.

Exercici 8 *Definir una funció per a* seleccionar *els elements parells d'una llista d'enters:*

```
selectEven :: [Int] -> [Int]
```

Per exemple, l'avaluació de l'expressió:

```
> selectEven [1,2,4,5,8,9,10]
```

retorna la llista `[2,4,8,10]`.

Exercici 9 *Definir una funció per a* seleccionar *els elements que ocupen les "posicions parells" d'una llista d'enters (recorda que les posicions en una llista comencen pel l'ndex zero, seguint el funcionament de l'operador* `!!`*):*

```
selectEvenPos :: [Int] -> [Int]
```

Per exemple, l'avaluació de l'expressió:

```
> selectEvenPos [1,2,4,5,8,9,10]
```

retorna la llista `[1,4,8,10]`.

Exercici 10 *Definir una funció* `iSort` *per a ordenar una llista en sentit ascendent. Per a açò, definir abans una funció* `ins` *que inserisca correctament un element en una llista ordenada (l'ordenació es pot resoldre recursivament considerant successives operacions d'inserció dels elements a ordenar en la part de la llista ja ordenada):*

```
iSort :: [Int] -> [Int]
ins :: Int -> [Int] -> [Int]
```

Per exemple, l'avaluació de l'expressió:

```
> iSort [4,9,1,3,6,8,7,0]
```

retorna la llista `[0,1,3,4,6,7,8,9]`. *I l'avaluació de l'expressió:*

```
> ins 5 [0,1,3,4,6,7,8,9]
```

retorna la llista `[0,1,3,4,5,6,7,8,9]`.

Les cadenes de caràcters vistes en la pràctica anterior (per exemple, `"hello"`) són simples llistes de caràcters escrites amb una sintaxi especial, que omet les cometes simples. Així, `"hello"` és tan sol una sintaxi convenient per a la llista `['h','e','l','l','o']`. Tota operació genèrica sobre llistes és, per tant, aplicable també a les cadenes.

4.2 Els rangs

Ja hem vist els rangs en alguns exemples previs. La forma bàsica té la sintaxi `[first..last]` de manera que genera la llista de valors entre tots dos (inclusivament):

```
> [10..15]
[10,11,12,13,14,15]
```

La sintaxi dels rangs en Haskell permet les següents opcions:

- `[first..]`
- `[first,second..]`
- `[first..last]`
- `[first,second..last]`

El comportament Del qual es pot deduir dels següents exemples:

```
[0..] -> 0, 1, 2, 3, 4, ...
[0,10..] -> 0, 10, 20, 30, ...
[0,10..50] -> 0, 10, 20, 30, 40, 50
[10,10..] -> 10, 10, 10, 10, ...
[10,9..1] -> 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
[1,0..] -> 1, 0, -1, -2, -3, ...
[2,0..(-10)] -> 2, 0, -2, -4, -6, -8, -10
```

4.3 Les llistes intensionales

Haskell proporciona una notació alternativa per a les llistes, les nomenades *llistes intensionales* (notació que és útil també per a descriure càlculs que necessiten `map` i `filter`, tal com es veurà en la següent secció). Ací teniu un exemple:

```
> [x * x | x <- [1..5], odd x]
[1,9,25]
```

L'expressió es llig: la llista dels quadrats dels nombres imparells en el rang d'1 a 5.

Formalment, una llista intensional és de la forma `[e|Q]`, on `e` és una expressió i `Q` és un *cualificador*. Un cualificador és una seqüència, potencialment buida, de *generadors* i *guardes* separats per comes. Un generador pren la forma `x <- xs`, on `x` és una variable o tupla de variables, i `xs` és una expressió de tipus llista. Una guarda és una expressió booleana. El cualificador `Q` pot ser buit, en aquest cas s'escriu simplement `[e]`. Ací teniu alguns exemples:

```
> [(a,b) | a <- [1..3], b <- [1..2]]
[(1,1),(1,2),(2,1),(2,2),(3,1),(3,2)]
>
> [(a,b) | a <- [1..2], b <- [1..3]]
[(1,1),(1,2),(1,3),(2,1),(2,2),(2,3)]
```

Els generadors posteriors poden dependre de variables introduïdes pels precedents, com en:

```
> [(i,j) | i <- [1..4], j <- [i+1..4]]
[(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]
```

Es poden intercalar lliurement generadors i guardes:

```
> [(i,j) | i <- [1..4], even i, j <- [i+1..4], odd j]
[(2,3)]
```

5 Les funcions `map` i `filter`

Es tracta de dues funcions predefinides útils per a operar amb llistes. La funció `map` aplica una funció a cada element d'una llista. És una típica funció de les denominades *d'ordre superior*, en acceptar, com a primer argument, no valors qualssevol sinó *funcions*. Per exemple:

```
> map square [9,3]
[81,9]
> map (<3) [1,2,3]
[True,True,False]
```

on `square` és la funció que calcula el quadrat d'un nombre enter (la qual podria definir-se, per exemple, mitjançant `let square = (^2)`). La definició de `map` és:

```
map          :: (a> - b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

Existeixen bastant lleis algebraiques útils relacionades amb `map`. Dos fets

bàsics són:

$$\texttt{map}\ id = id$$
$$\texttt{map}\ (f\ \cdot\ g) = (\texttt{map}\ f)\ \cdot\ (\texttt{map}\ g)$$

on *id* és la funció identitat i "$\cdot$" és la composició de funcions. La primera equació expressa que en aplicar la funció identitat a tots els elements d'una llista, la llista queda sense modificar. Les dues aparicions de `id` en la primera equació tenen tipus diferents: el de l'esquerra és *id* `::  a -> a` i el de la dreta *id* `::  [a] -> [a]`. La segona equació expressa que aplicar primer g a tot element d'una llista i després aplicar f a tot element de la llista resultat, dóna el mateix resultat que aplicar $f \cdot g$ a la llista original. Llegida de dreta a esquerra, aquesta llei permet reemplaçar dos recorreguts d'una llista per un solament, amb el corresponent guany en eficiència, per la qual cosa s'aconsella usar-la sempre que es puga.

Exercici 11 *Definir, usant la funció* `map`*, una funció per a duplicar tots els elements d'una llista d'enters:*

```
doubleAll :: [Int] -> [Int]
```

Per exemple, l'avaluació de l'expressió:

```
> doubleAll [1,2,4,5]
```

retorna la llista `[2,4,8,10]`*.*

La funció `filter` pren una funció booleana `p` i una llista `xs` i retorna la sublista de `xs` els elements del qual satisfan `p`. Per exemple:

```
> filter even [1,2,4,5,32]
[2,4,32]
```

La definició de `filter` és:

```
filter          :: (a> - Bool) -> [a] -> [a]
filter p []      = []
filter p (x:xs) = if p x then x:filter p xs else filter p xs
```

Exercici 12 *Expressar mitjançant llistes intensionales les definicions de les funcions* `map` *i* `filter`*. Nota: les pots nomenar* `map'` *i* `filter'` *per a evitar conflictes amb les funcions predefinides del* `Prelude`*.*

Exercici 13 *¿Què computa l'expressió misteriosa que apareix a continuació? (on* `sum` *és la funció predefinida que suma els elements d'una llista d'enters):*

```
> (sum . map square . filter even) [1..10]
```

6 Tipus algebraics

6.1 Enumeracions

En un llenguatge funcional com Haskell, el programador pot definir nous tipus amb els seus valors associats, emprant els anomenats tipus algebraics. Per exemple, amb la declaració:

```
data Color = Red | Green | Blue
```

s'estableix un nou tipus de dades `Color`. El tipus `Color` conté només tres valors o dades denotades pels corresponents constructors de dades constants `Red`, `Green` i `Blue`. Recorde's que, en Haskell, els identificadors que es refereixen a tipus de dades i a constructors de dades sempre comencen per una *lletra majúscula*.

6.2 Tipus Renomenat

També es poden declarar renomenaments de tipus ja definits, denominats tipus "sinònims". Per exemple:

```
type Distance = Float
type Angle = Float
type Position = (Distance,Angle)
type Pairs a = (a,a)
type Coordinates = Pairs Distance
```

De fet, el tipus `String` és un renomenament, com ja s'había esmentat abans:

```
type String = [Char]
```

i se li pot aplicar totes les operacions sobre llistes, per exemple les comparacions d'ordre (ordre lexicogràfic):

```
> "hola" < "halo"
False
> "ho" < "hola"
True
```

Sovint es prefereix mostrar les cadenes sense que apareguen les dobles cometes en l'eixida i on els caràcters especials, com a salt de línia, etc., s'imprimisquen com el caracter real que representen. Haskell disposa de *ordres* primitives per a imprimir cadenes, llegir de fitxers, etc. Per exemple, usant la funció `chr` que converteix un enter al caràcter que aquest representa:

```
> putStr ("Aquesta frase té un" ++ [chr 10] ++ "fi de linea")
Aquesta frase té una
fi de linea
```

Es recorda la necessitat d'importar el mòdul `Data.Char` per a usar la funció `chr`.

Exercici 14 *Definir els següents tipus "sinònims":*

```
type Person = String
type Book = String
type Database = [(Person,Book)]
```

El tipus `Database` *defineix una base de dades d'una biblioteca com una llista de parells* `(Person,Book)` *on* `Person` *és el nom de la persona que té en préstec el llibre* `Book`*. Un exemple de base de dades és:*

```
exampleBase :: Database
exampleBase = [("Alicia","El nombre de la rosa"),("Juan",
 "La hija del canibal"),("Pepe","Odesa"),("Alicia",
 "La ciudad de las bestias")]
```

A partir d'aquesta base de dades exemple es poden definir funcions per a obtenir els llibres que té en préstec una persona donada, `obtain`*, per a realitzar un préstec,* `borrow`*, i per a realitzar una devolució,* `return'`*.*
Per exemple, la funció `obtain` *es pot definir així:*

```
obtain :: Database -> Person -> [Book]
obtain dBase thisPerson
 = [book | (person,book) <- dBase, person == thisPerson]
```

que significa que la funció retorna la llista de tots els llibres tals que hi ha un parell `(person,book)` *en la base de dades i* `person` *és igual a la persona els llibres de la qual s'està cercant. Per exemple, l'avaluació de l'expressió:*
`obtain exampleBase "Alicia"`
retorna la llista: `["El nombre de la rosa","La ciudad de las bestias"]`

Completar el programa amb les definicions per a les funcions `borrow` *i* `return'`*:*

```
borrow :: Database -> Book -> Person -> DataBase
return' :: Database -> (Person,Book) -> DataBase
```

6.3 Arbres

La declaració:

```
data TreeInt = Leaf Int | Branch TreeInt TreeInt
```

estableix un nou tipus de dades `TreeInt` (on `TreeInt` és, de nou, un constructor de tipus constant) que consta d'un nombre infinit de valors definits recursivamente amb l'ajuda dels símbols constructors de dades `Leaf` (unario) i `Branch` (binari), que prenen com a arguments un nombre enter i dos arbres `TreeInt`, respectivament.

Consideren-se alguns exemples de valors o dades dels tipus anteriors:

- `[Red,Green,Red]` és un valor de tipus `[Color]`
- `Branch (Leaf 0) (Branch (Leaf 0) (Leaf 1))` és un valor de tipus `TreeInt`.

Gens impedeix definir tipus genèrics emprant aquests recursos expressius. El tipus

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

és un tipus algebraic polimòrfic per a definir arbres de qualsevol tipus de dades, de forma similar a com les llistes admeten qualsevol tipus de dades.

Per descomptat, també es poden definir funcions sobre aquests nous tipus de dades algebraics definits per l'usuari. La funció `numleaves` definida com segueix:

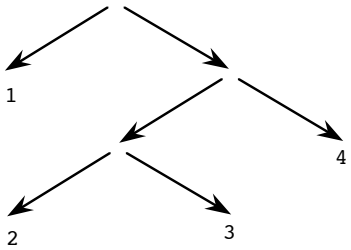
```
numleaves (Leaf x)     = 1
numleaves (Branch a b) = numleaves a + numleaves b
```

calcula el nombre de fulles d'un arbre del tipus `Tree a`

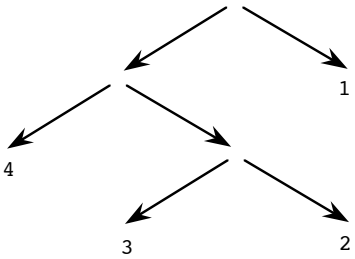
Exercici 15 *Escriure la definició de la funció* `numleaves` *en un fitxer i carregar-ho en l'intèrpret* GHCi *per a activar la definició de la funció. Establir l'expressió de tipificació per a la funció* `numleaves` *i consultar la proporcionada per l'intèrpret.*

Exercici 16 *Definir una funció que obtinga l'arbre simètric al que se li passa com a paràmetre.*

```
symmetric :: Tree a> - Tree a
```



Un árbol de enteros...



...y su simétrico

Nota: Si es prova la funció `symmetric` amb `ghci` es vorà que dóna un error:

```
> symmetric (Branch (Leaf 5) (Leaf 7))
<interactive>:5:1:
    No instance for (Show (Tree a0))
      arising from a use of 'print'
    Possible fix: add an instance declaration for (Show (Tree a0))
    In a stmt of an interactive GHCi command: print it
```

açò és a causa que no sap com mostrar el resultat. El resultat es mostra amb la funció `show` (en certa manera, com el de `toString` Java). va a utilitzar-se un mecanisme molt senzill per a proporcionar un comportament per defecte per a mostrar, amb `show`, un tipus de dades algebraic. N'hi

ha prou amb afegir `deriving Show` en finalitzar la declaració d'un tipus de dades algebraic:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a) deriving Show
```

Exercici 17 *Definir les funcions*

```
listToTree :: [a] -> Tree a
treeToList :: Tree a -> [a]
```

la primera de les quals converteix una llista no buida en un arbre, realitzant la segona d'elles la transformació contrària.

Considere's ara la següent definició alternativa per al tipus de dades dels arbres binaris d'enters

```
data BinTreeInt  =
Void | Node Int BinTreeInt BinTreeInt deriving Show
```

en el qual els valors sencers s'emmagatzemen en els nodes i on les fulles són subarbres que tenen tan sol arrel (denotats pel símbol constructor de dades `Void`). A continuació, es mostren alguns exemples d'arbres binaris d'enters:

```
let treeB1 = Void
let treeB2 = (Node 5) Void Void
let treeB3 = (Node 5)
              ((Node 3)(Node 1 Void Void)(Node 4 Void Void))
              ((Node 6) Void (Node 8 Void Void))
```

on `treeB1` és un arbre buit, `treeB2` és un arbre amb un sol element, i `treeB3` és un arbre amb el valor 5 en el seu node arrel, els valors 3, 1, 4 en els nodes de la seua branca esquerra, i els valors 6, 8 en els nodes de la seua branca dreta.

Es diu que un arbre binari està ordenat si els valors emmagatzemats en el subarbre esquerre d'un node donat són sempre menors o iguals al valor en aquest node, mentre que els valors emmagatzemats en el seu subarbre dret són sempre majors. Aquest tipus d'arbres també rep el nom d'arbre binari de cerca (*binary search tree*). Els anteriors exemples corresponen a arbres ordenats.

Sobre aquest tipus de dades resolguen-se els següents exercicis.

Exercici 18 *Definir una funció*

```
insTree :: Int ->  BinTreeInt -> BinTreeInt
```

per a inserir un valor sencer en el seu lloc en un arbre binari ordenat.

Exercici 19 *Donada una llista no ordenada d'enters, definir una funció*

```
creaTree :: [Int] ->  BinTreeInt
```

que construïsca un arbre binari ordenat a partir de la mateixa.

Exercici 20 *Definir una funció*

```
treeElem :: Int ->  BinTreeInt -> Bool
```

que determine "de forma eficient" si un valor sencer pertany o no a un arbre binari ordenat.

Exercici 21 *Considere's l'anterior declaració d'arbre binari d'enters:*

```
data BinTreeInt  = Void | Node Int BinTreeInt BinTreeInt deriving Show
```

Es vol implementar una funció `dupElem` *que retorne un arbre amb la mateixa estructura però amb tots els seus valors duplicats.*

Considere's que s'aplicara `dupElem` *als arbres* `treeB1`, `treeB2` *i* `treeB3` *declarats anteriorment.*
L'avaluació de l'expressió:

```
> dupElem treeB1
```

retorna `Void`. *L'avaluació de l'expressió:*

```
> dupElem treeB2
```

retorna `Node 10 Void Void`. *I l'avaluació de l'expressió:*

```
> dupElem treeB3
```

retorna:

```
Node 10
(Node 6 (Node 2 Void Void) (Node 8 Void Void))
(Node 12 Void (Node 16 Void Void)).
```

Exercici 22 *Considere's la següent declaració:*

```
data Tree a = Branch a (Tree a) (Tree a) | Void deriving Show
```

Es vol implementar una funció `countProperty` *amb la següent signatura:*

```
countProperty :: (a> - Bool) -> (Tree a) -> Int
```

que retorne el nombre d'elements de l'arbre que compleixen la propietat.
Per exemple, l'avaluació de l'expressió:

```
> countProperty (>9) (Branch 5 (Branch 12 Void Void) Void)
```

retorna `1`, *i l'avaluació de l'expressió:*

```
> countProperty (>0) (Branch 5 (Branch 12 Void Void) Void)
```

retorna `2`.

7 Avaluació

L'assistència a les sessions de pràctiques és obligatòria per a aprovar l'assignatura. L'avaluació d'aquesta segona part de pràctiques es realitzarà mitjançant un examen individual en el laboratori.