Tipos enumerados

Definimos el tipo enumerado `Animal` de la siguiente forma:

```
data Animal = Ardilla | Buho | Cerdo | Delfin | Erizo | Foca |
    Ganso | Halcon | Iguana
```

Podemos preguntar por el tipo de un elemento:

```
*Main> :t Buho
Buho :: Animal
```

Por defecto, Haskell no sabe "mostrar" un elemento de tipo `Animal`:

```
*Main> Erizo
<interactive>:24:1:
    No instance for (Show Animal) arising from a use of `print'
    Possible fix: add an instance declaration for (Show Animal)
    In a stmt of an interactive GHCi command: print it
```

Esto se arregla agregando `deriving (Show)` a la definición del tipo:

```
data Animal = Ardilla | Buho | Cerdo | Delfin | Erizo | Foca |
    Ganso | Halcon | Iguana   deriving (Show)
```

Ahora sí:

```
*Main> Erizo
Erizo
```

Tipos enumerados

```
data Animal = Ardilla | Buho | Cerdo | Delfin | Erizo | Foca |
    Ganso | Halcon | Iguana   deriving (Show)
```

Definamos una función que nos diga si un animal es un ave:

```
esAve :: Animal -> Bool
esAve Buho = True
esAve Ganso = True
esAve Halcon = True
esAve _ = False
```

```
*Main> esAve Delfin
False
*Main> esAve Halcon
True
```

Otra definición posible:

```
esAve2 :: Animal -> Bool
esAve2 x  =  x==Buho || x==Ganso || x==Halcon
```

Pero esto tira error: `No instance for (Eq Animal) arising from a use of '=='`. Es decir, no está definida la igualdad entre animales. Para tipos enumerados, podemos arreglarlo agregando `Eq` además de `Show` a la definición del tipo:

```
data Animal = Ardilla | Buho | Cerdo | Delfin | Erizo | Foca |
    Ganso | Halcon | Iguana   deriving (Show,Eq)
```

Tipos enumerados

```
data Animal = Ardilla | Buho | Cerdo | Delfin | Erizo | Foca |
    Ganso | Halcon | Iguana   deriving (Show,Eq)
```

Ahora podemos comparar evaluar la igualdad de elementos de tipo Animal:

```
*Main> Iguana == Ardilla
False
*Main> Cerdo == Cerdo
True
*Main> Foca /= Foca
False
```

Como con cualquier otro tipo, podemos definir listas de animales:

```
misFavoritos :: [Animal]
misFavoritos = [Delfin, Buho, Ardilla, Halcon, Foca]
```

y funciones para listas:

```
primeraAve :: [Animal] -> Animal
primeraAve (x:xs) | esAve x = x
                  | otherwise = primeraAve xs
```

```
*Main> primeraAve misFavoritos
Buho
```

Listas infinitas

Ya vimos cómo definir rangos en Haskell:

```
*Main> [1..10]
[1,2,3,4,5,6,7,8,9,10]
```

Cuando no especificamos un límite superior, obtenemos una lista infinita.

```
*Main> [1..]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,
26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,
48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,
70,71,72,73,74,75,76,77,78,79,80,81,82,83,84,85,...
```

Además, podemos definir funciones para crear listas infinitas:

```
todosLosNaturalesDesde :: Integer -> [Integer]
todosLosNaturalesDesde n = n:todosLosNaturalesDesde (n+1)
```

```
*Main> todosLosNaturalesDesde 1022
[1022,1023,1024,1025,1026,1027,1028,1029,1030,1031,1032,1033,
1034,1035,1036,1037,1038,1039,1040,1041,1042,1043,1044,1045,
1046,1047,1048,1049,1050,1051,1052,1053,1054,1055,1056,1057...
```

Listas infinitas

```
todosLosNaturalesDesde :: Integer -> [Integer]
todosLosNaturalesDesde n = n:todosLosNaturalesDesde (n+1)
```

En general, siempre esperamos que un cómputo sea finito. Para lograrlo, al trabajar con una lista infinita, trabajaremos solo con una porción finita de ella. Haskell nos provee la función take, que dados un natural *n* y una lista *l*, devuelve los primeros *n* elementos de *l*.

```
take :: Int -> [a] -> [a]
```

Usando `take`, y aprovechando que Haskell es *lazy*, podemos obtener prefijos finitos de listas infinitas:

```
*Main> take 10 (todosLosNaturalesDesde 1022)
[1022,1023,1024,1025,1026,1027,1028,1029,1030,1031]
```