Lección 01

Primer Contacto

El contacto incial con la consola de Haskell podría ser según lo siguiente:

```
User:lec_01$ ghci
GHCi, version 8.0.2: http://www.haskell.org/ghc/  :? for help
Prelude> 2 + 15
17
Prelude> 2-15
-13
Prelude> 2*3
6
Prelude> 5/2
2.5
Prelude> True && False
False
Prelude> True && True
True
Prelude> False || True
True
Prelude> not False
True
Prelude> not (True && False)
True
Prelude> 5 == 5
True
Prelude> 5 == 3+2
True
Prelude> 5 /= 4
True
Prelude> 5 /= 5
False
Prelude> "hola" == "hola"
True
Prelude> 'hola'

<interactive>:15:1: error:
    • Syntax error on 'hola'
      Perhaps you intended to use TemplateHaskell or TemplateHaskellQuotes
     • In the Template Haskell quotation 'hola'
Prelude> 'h'
'h'
Prelude> succ 8
9
Prelude> succ (-1)
0
Prelude> min 9 (-3)
-3
Prelude> max 9 (-3)
```

```
9
Prelude> suc 9 + max 5 4 +1

<interactive>:21:1: error:
    • Variable not in scope: suc :: Integer -> a
    • Perhaps you meant one of these:
      ‘sum’ (imported from Prelude), ‘succ’ (imported from Prelude)
Prelude> succ 9 + max 5 4 +1
16
Prelude> (succ 9) + (max 5 4) +1
16
Prelude> (succ -9) + (max 5 4) +1

<interactive>:24:1: error:
    • Non type-variable argument in the constraint: Ord (a -> a)
      (Use FlexibleContexts to permit this)
    • When checking the inferred type
      it :: forall a. (Ord (a -> a), Enum a, Num (a -> a)) => a -> a
Prelude> (succ (-9)) + (max 5 4) +1
-2
Prelude> 'U':"n gato negro"
"Un gato negro"
Prelude> "Un gato negro" !! 3
'g'
Prelude> "Un gato negro" !! 2
' '
Prelude> let n = 2::Integer
Prelude> "Un gato negro" !! n

<interactive>:30:20: error:
    • Couldn't match expected type ‘Int’ with actual type ‘Integer’
    • In the second argument of ‘(!!)’, namely ‘n’
      In the expression: "Un gato negro" !! n
      In an equation for ‘it’: it = "Un gato negro" !! n
Prelude> let lst = [5,4,3,2,1,0]
Prelude> head lst
5
Prelude> tail lst
[4,3,2,1,0]
Prelude> lst == (head lst):(tail lst)
True
Prelude> last lst
0
Prelude> init lst
[5,4,3,2,1]
Prelude> lst == (init lst)++[last lst]
True
Prelude> lst == reverse ((last lst):reverse (init lst))
True
Prelude> head []
*** Exception: Prelude.head: empty list
Prelude> length lst
```

```
6
Prelude> null []
True
Prelude> null lst
False
Prelude> null ""
True
Prelude> null " "
False
Prelude> take 3 lst
[5,4,3]
Prelude> take 3 []
[]
Prelude> take (-1) []
[]
Prelude> take (-1) lst
[]
Prelude> null (take (-1) lst)
True
Prelude> lst
[5,4,3,2,1,0]
Prelude> drop 3 lst
[2,1,0]
Prelude>
Prelude> drop 0 lst
[5,4,3,2,1,0]
Prelude> drop 10 lst
[]
Prelude> maximum lst
5
Prelude> minimum lst
0
Prelude> sum lst
15
Prelude> lst
[5,4,3,2,1,0]
Prelude> product lst
0
Prelude> elem 4 lst
True
Prelude> 4 `elem` lst
True
Prelude> (-1) `elem` lst
False
Prelude> [1..20]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
Prelude> [2, 4..20]
[2,4,6,8,10,12,14,16,18,20]
Prelude> [2,4,8..64]

<interactive>:71:7: error: parse error on input ‘..’
Prelude> [3,6..20]
```

```
[3,6,9,12,15,18]
Prelude> [0.1,0.3..1]
[0.1,0.3,0.5,0.7,0.8999999999999999,1.0999999999999999]
Prelude> take 10 (cycle lst)
[5,4,3,2,1,0,5,4,3,2]
Prelude>
Prelude> take 10 (repeat 0)
[0,0,0,0,0,0,0,0,0,0]
Prelude> take 10 (repeat [0,1])
[[0,1],[0,1],[0,1],[0,1],[0,1],[0,1],[0,1],[0,1],[0,1],[0,1]]
Prelude> take 10 (repeat "abc")
["abc","abc","abc","abc","abc","abc","abc","abc","abc","abc"]
Prelude> replicate 3 10
[10,10,10]
Prelude> replicate 3 "abc"
["abc","abc","abc"]
Prelude> [x*2 | x<-[1..10]]
[2,4,6,8,10,12,14,16,18,20]
Prelude> [x | x<-[50..100], x `mod` 7 == 0]
[56,63,70,77,84,91,98]
Prelude> [x | x <- [10..20], x /= 13, x /= 15, x /= 19]
[10,11,12,14,16,17,18,20]
Prelude> [ x*y | x <- [2,5,10], y <- [8,10,11]]
[16,20,22,40,50,55,80,100,110]
Prelude> [ x*y | x <- [2,5,10], y <- [8,10,11], x*y > 50]
[55,80,100,110]
Prelude> [1 | _ <- lst]
[1,1,1,1,1,1]
Prelude> sum [1 | _ <- lst]
6
Prelude> length lst
6
Prelude> [ (x,y) | x <- [2,5,10], y <- [8,10,11], x+y==5]
[]
Prelude> [ (x,y) | x <- [2,5 .. 100], y <- [8,10..110], x+y==5]
[]
Prelude> [ (x,y) | x <- [0..100], y <- [0..110], x+y==5]
[(0,5),(1,4),(2,3),(3,2),(4,1),(5,0)]
Prelude> fst (0,5)
0
Prelude> snd (0,5)
5
Prelude> let t@(x,y,z)=(-1,0,1)
Prelude> t
(-1,0,1)
Prelude> x
-1
Prelude> y
0
Prelude> z
1
Prelude> removeNonUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]
```

```
Prelude> removeNonUppercase "un gato negro"
""
Prelude> removeNonUppercase "Un gAtO nEgrO"
"UAOEO"
Prelude> zip [1,2,3,4] ['a','b','c']
[(1,'a'),(2,'b'),(3,'c')]
Prelude> let triangulos = [ (a,b,c) | c <- [1..10], b <- [1..10], a <- [1..10] ]
Prelude> let triangulosBuenos = [ (a,b,c) | c <- [1..10], b <- [1..c], a <- [1..b],
a^2 + b^2 == c^2]
Prelude> rightTriangles
[(3,4,5),(6,8,10)]
Prelude>
```

Definición de Funciones por Casos

Los comentarios en el código `Haskell` se incluye con el código:

```
-- línea comentada
```

o bien con el código:

```
{-
Este texto, que ocupa un parrafo completo,
queda comentado
-}
```

La forma más simple de definir funciones en `Haskell` es hacerlo por casos:

```
suerte :: (Integral a) => a -> String
suerte 7 = "¡El siete de la suerte!"
suerte _ = "Lo siento, ¡no es tu día de suerte!"
```

En el anterior código la función actua sobre cualquier tipo de dato `a` a condición de estar dicho tipo incluido en la clase `Integral`. Determinamos un comportamiento de la función para el patrón coincidente con `7` y otro para el resto. Así `_` debe ser entendido como `en otro caso`. En general recurriremos a esta "denominación anónima" siempre que la actuación no requiera ningún rasgo del patrón por la que es invocada.

En este tipo de definiciones el orden de las líneas suele importar, hasta el extremo de que tal orden puede marcar la diferencia entre un código que funciona y otro que no. En efecto, si `Prueba.hs` fuese un fichero con este contenido:

```
module Prueba (module Prueba) where

fac :: Integer -> Integer
fac 0 = 1
fac n = n * fac (n-1)

facM :: Integer -> Integer
facM n = n * facM (n-1)
facM 0 = 1
```

generaría el siguiente diálogo:

```
*Prelude> :l Prueba
[1 of 1] Compiling Prueba           ( Prueba.hs, interpreted )

Prueba.hs:10:1: warning: [-Woverlapping-patterns]
    Pattern match is redundant
    In an equation for ‘facM’: facM 0 = ...
    |
 10 | facM 0 = 1
    | ^^^^^^^^^^
Ok, one module loaded.
```

indicando que al analizar la función `facM` se ha encontrado una redundancia en la distribución de casos según patrones; no obstante, tras la advertencia, el módulo es cargado. Sin embargo al intentar calcular obtenemos el siguiente diálogo:

```
*Prueba> fac 13
6227020800
*Prueba> facM 13
^CInterrupted.
*Prueba>
```

en el que queda patente que `facM 13` fracasa al no encontrar condición de parada. Tenga en cuenta que al definir por casos, es ejecutada la función según el código de la primera línea en la que el dato ajusta con el patrón distintivo de la misma según la división de casos.

Hay casos en los que es indiferente el orden de las líneas. Como ejemplo considere el siguiente:

```
cabeza :: [a] -> a
cabeza [] = error "la lista es vacía"
cabeza (x:_) = x

cabezaAlt :: [a] -> a
cabezaAlt (x:_) = x
cabezaAlt [] = error "la lista es vacía"
```

Considere y explíquese los siguientes ejemplos:

```
charName :: Char -> String
charName 'a' = "Alberto"
charName 'b' = "Benito"
charName _ = "Tu mismo"

longitud :: (Num b) => [a] -> b
longitud [] = 0
longitud (_:xs) = 1 + longitud xs

sumarVectores :: (Num a) => (a,a) -> (a,a) -> (a,a)
sumarVectores (x,y) (a,b) = (s x a,s y b)
                          where
                           s m n = m + n
```

y especialmente explíquese el sentido de la sintaxis `where`.

En el siguiente ejemplo abundamos en la definición por casos, pero destacamos el interés particular de su cuarta línea de código:

```
lst :: Int -> [Int]
lst 0 = []
lst 1 = [1]
lst 2 = [2*i | i <- [0..10]]
lst n = (lst (n-2)) ++ (lst 2)
```

en el que la definición es una lista por compresión en la que son seleccionados exactamente los números que son el doble de los que figuran en la lista de las instancias de la clase `Int` comprendidas entre `0` y `10`, ambos incluidos. La quinta línea comporta una concatenación una vez hecha una doble llamada recursiva a la propia función `lst`. Es un mal estilo de concatenar que muchas veces puede ser salvado, lo cual será ensayado más adelante.

Es posible incluir un pseudónimo en el patrón el uso de un pseudónimo de lo que sea ajustado al mismo:

```
capital :: String -> String
capital "" = error "el string vacío no tiene primer caracter"
capital all@(x:_) = "la primera letra de " ++ all ++ " es " ++ [x]
```

dicho código ofrece el siguiente diálogo:

```
Prelude> :l Prueba
[1 of 1] Compiling Prueba           ( Prueba.hs, interpreted )
Ok, one module loaded.
*Prueba> capital ""
"*** Exception: el string vacío no tiene primer caracter
CallStack (from HasCallStack):
error, called at Prueba.hs:32:14 in main:Prueba
*Prueba> capital "abc"
"la primera letra de abc es a"
*Prueba>
```

Ahora vemos que si los corchetes indican lista, las comillas doble indican lista de caracteres. En `Haskell` `['a','b','c']` es sinónimo de `"abc"`.

Otra forma de definir funciones es hacerlo por condiciones en lugar de por patrones:

```
calificacion :: (RealFloat a) => a -> String

calificacion nota
     | nota <= 4.99             = "Suspenso"
     | nota >= 5 && nota < 6    = "Aprobado"
     | nota >= 6 && nota < 7    = "Bien"
     | nota >= 7 && nota < 9    = "Notable"
     | nota >= 9 && nota < 10   = "Sobresaliente"
     | nota >= 10               = "MH"
```

En las anteriores construcciones booleanas vemos el uso de la conectiva `&&` que tiene el sentido de `and` en otros lenguajes.

Es posible usar respuestas con los valores predeterminados de la clase `Ord`:

```
miComparador :: (Ord a) => a -> a -> Ordering
a `miComparador` b
    | a > b     = GT
    | a == b    = EQ
    | otherwise = LT
```

retazo de código que proporciona el siguiente diálogo:

```
Prelude> :l Prueba
[1 of 1] Compiling Prueba           ( Prueba.hs, interpreted )
Ok, one module loaded.
*Prueba> 3 `myCompare` 5
LT
*Prueba> myCompare 3 5
LT
*Prueba>
```

En el código anterior se muestra como es posible usar de forma prefija e infija una función definida. De todas formas también podemos definir una función como una operación infija, asignándole entre otros una precedencia:

```
infixl 9 !!!
(!!!) :: [a] -> Integer -> a
(x:_)  !!! 0         = x
(_:xs) !!! n | n > 0 = xs !!! (n-1)
(_:_)  !!! _         = error "negative index"
[]     !!! _         = error "too large index"
```

Otra sintaxis para conseguir definir por casos es a través de `case`:

```
tomar :: Int -> [a] -> [a]
tomar m lst = case (m,lst) of
               (0,_)    -> []
               (_,[])   -> []
               (n,x:xs) -> f n (x:xs)
             where
              f :: Int -> [a] -> [a]
              f n (x:xs)
                | n < 0 = []
                | otherwise = x : tomar (n-1) xs
```

Esta función toma las `n` primeras entradas de la lista que es su segundo argumento.

Más ejemplos con la sintaxis `case` son los siguientes:

```
head' :: [a] -> a
head' xs = case xs of [] -> error "No head for empty lists!"
                      (x:_) -> x
```

es la versión de:

```
head' :: [a] -> a
head' [] = error "No head for empty lists!"
head' (x:_) = x
```

La sintaxis general de `case` es:

```
case expression of pattern -> result
                   pattern -> result
                   ...
                   pattern -> result
```

donde `expression` es tratada de casar contra los patrones `pattern`. La acción de ajuste de patrones se comporta como se espera: el primer patrón que se ajuste es el que se utiliza. Si no se puede ajustar a ningún patrón de la expresión case se lanzará un error de ejecución.

Mientras que el ajuste de patrones de los parámetros de una función puede ser realizado únicamente al definir una función, las expresiones case pueden ser utilizadas casi en cualquier lugar. Por ejemplo considérese el siguiente elegantísimo código:

```
describeList :: [a] -> String
describeList xs = "The list is " ++ case xs of [] -> "empty."
                                               [x] -> "a singleton list."
                                               xs -> "a longer list."
```

Son útiles para realizar un ajuste de patrones en medio de una expresión. Como el ajuste de patrones que se realiza en la definición de una función es una alternativa sintáctica a las expresiones `case`, también podríamos utilizar algo como esto:

```
describeList :: [a] -> String
describeList xs = "The list is " ++ what xs
    where what [] = "empty."
          what [x] = "a singleton list."
          what xs = "a longer list."
```

Seguidamente implementamos una simulación de la función `reverse`, que invierte una lista, de varias formas con muy distinta eficiencia:

```
revertirM :: [a] -> [a]
revertirM [] = []
revertirM (cabeza:cola) = (revertirM cola) ++ [cabeza]

revertirAux :: [a] -> [a] -> [a]
revertirAux lst [] = lst
revertirAux lst (x:xs) = revertirAux  (x:lst) xs

revertir :: [a] -> [a]
revertir = revertirAux []
```

```
revertirBM :: [a] -> [a]
revertirBM lst = [lst !! i | i <- [length(lst)-1,length(lst)-2 .. 0]]
```

Seguidamente tenemos una simulación de la función `replicate`:

```
replicate' :: Int -> a -> [a]
replicate' n x
      | n == 0    = []
      | otherwise = x: replicate' (n-1) x
```

pero tenga cuidado, una vez más la alteración del orden de las líneas puede tener fatales consecuencias:

```
replicate' :: Int -> a -> [a]
replicate' n x
      | otherwise = x: replicate' (n-1) x
      | n == 0    = []
```

Ejercicios

1. Defina una función sobre listas de números, digamos `sLst` que ofrezca el siguiente diálogo:

   ```
   *Prueba> sLst []
   0
   *Prueba> sLst [1,2]
   1
   *Prueba> sLst [1,2,3]
   3
   *Prueba> sLst [1,2,3,4]
   6
   *Prueba> sLst [-1,1,2,3,4]
   4
   ```

   y ello a modo de ilustración de cómo ha de funcionar sobre cualquier lista finita.

2. Diseñe una función que emita un mensaje por pantalla en función de la relación entre el peso y el cuadrado de la altura. Digamos que tenga tres umbrales definidos y el resto.

3. Explique el sentido de cada línea de código del retazo definitorio de la operación infija `!!!`.

4. Investigue el funcionamiento de las funciones `divMod` y `quotRem`, explíqueselo y simule con una implementación propia su funcionamiento.

5. Defina una función, de nombre `producto` digamos, que multiplique los elementos de una lista de instancias de la clase `Num`.

6. Analice el funcionamiento de la función primitiva `drop` y de una implementación propia.

7. Repita el ejercicio anterior para las funciones: `length`, `elem`, `head`, `take`, `init`, `tail` y `maximum`.

8. Explique el sentido y funcionamiento de la función del siguiente retazo de código:

```
boomBangs :: [Int] -> [String]
boomBangs xs = [if x <10 then "BOOM!" else "BANG!" | x <- xs, odd x]
```

9. Implemente una función que se aplique a listas y propocione un mensaje descriptivo acorde en función de que sea: vacía, unitaria o no esté en ninguno de esos casos.