Programación Declarativa – Haskell – Informática Sistemas – Curso 2008-2009
Pepe Gallardo – Universidad de Málaga

Tema 1

1.1 Considérense las siguientes definiciones de funciones:

$$\begin{array}{lcl} inc & :: & Float \rightarrow Float \\ inc\ x & = & x + 1.0 \end{array}$$

$$\begin{array}{lcl} f & :: & Float \rightarrow Float \rightarrow Float \\ f\ x\ y & = & x + (4.0 * x) \end{array}$$

Sabiendo que si se intenta reducir un expresión de la forma $x\ /0.0$, donde x es cualquier número real, se produce un error y la reducción acaba, reduce la expresión $f\ (inc\ 5.0)\ (7.0/0.0)$ utilizando orden aplicativo, orden normal y evaluación perezosa.

1.2 Sean las siguientes defincione

$$\begin{array}{lcl} doble & :: & Integer \rightarrow Integer \\ doble\ x & = & x + x \end{array}$$

$$\begin{array}{lcl} cuadruple & :: & Integer \rightarrow Integer \\ cuadruple\ x & = & doble\ (doble\ x) \end{array}$$

Reduce la expresión $cuadruple\ (1 + 2)$ utilizando orden aplicativo, orden normal y evaluación perezosa.

Tema 2

2.1 ¿Cuál es el tipo de las siguientes expresiones (en caso de que sean correctas)?

$(True,\ True)$
$(True,\ False)$
$(True,\ 'a',\ False)$
$(True,\ ('a',\ False))$
$(1 > 7,\ even\ 4,\ isUpper\ 'a')$
$['a',\ 'b',\ 'c']$
"abc"
$[True,\ 'a',\ False]$
$([True, False],\ 'a')$

Tema 3

3.1 ¿Cuáles de los siguientes patrones y argumentos unifican?. ¿Qué valores se asocian a cada variable en caso de que haya unificación?

Patrón	**Argumento**
0	0
x	0
$(x : ys)$	$[1, 2]$
$(x : ys)$	[”sam”]
$(x : ys)$	”sam”
$(1 : xs)$	$[1, 2]$
$(1 : xs)$	$[2, 3]$
$(x : \_ : \_ : ys)$	$[1, 2, 3, 4, 5, 6]$
$[\,]$	$[\,]$
$[\ x\ ]$	[”sam”]
$[x]$	[”sam”]
$[1, x]$	$[1, 2]$
$[x, y]$	$[1]$
$x@y$	0
$a@(x : b@(y : zs))$	$[1, 2, 3, 4]$
$(n + 2)$	6
$(n + 1)$	0

3.2 Escribe una función

$$descomponer \ :: \ Integer \ \rightarrow \ (Integer, \ Integer, \ Integer)$$

que a partir de una cantidad de segundos, devuelva las horas, minutos y segundos equivalentes. Da dos versiones, una usando **where** y otra usando **let in**.

3.3 Define una función que incremente todos los elementos de una tupla de tres enteros. Por ejemplo

? $incTupla3\ (1, 5, 8)$
$(2, 6, 9) \ :: \ (Integer, \ Integer, \ Integer)$

3.4 Define una función que incremente todos los elementos de una lista de enteros. Por ejemplo

? $incLista\ [1, 5, 8]$
$[2, 6, 9] \ :: \ [Integer]$

3.5 Sea una lista de funciones de enteros en enteros

$[f_1,\ f_2,\ \ldots\ f_n]\ \ ::\ [Integer\ \rightarrow\ Integer]$

Define un operador ($|>$) de forma que se tenga

$$[f_1,\ f_2,\ \ldots\ f_n]\ |>\ x\ \Longrightarrow\ [f_1\ x,\ f_2\ x,\ \ldots\ f_n\ x]$$

3.6 Escribe una función que determine si un año es bisiesto. Un año es bisiesto si es múltiplo de 4 (por ejemplo 1984). Una excepción a la regla anterior es que los años múltiplos de 100 sólo son bisiestos cuando a su vez son múltiplos de 400 (por ejemplo 1800 no es bisiesto, mientras que 2000 lo será):

? *esBisiesto* 1984
True :: *Bool*

3.7 Escribe una función recursiva que devuelva el resto de la división de dos enteros usando sustracciones.

3.8 Escribe una función recursiva que devuelva el cociente que se obtiene al dividir dos números enteros usando sumas y restas.

3.9 Escribe una función recursiva que devuelva el sumatorio desde un valor entero hasta otro:

$$sumDesdeHasta\ a\ b\ \Longrightarrow\ a\ +\ (a+1)\ +\ (a+2)\ +\ \ldots\ +\ (b-1)\ +\ b$$

3.10 Escribe una función recursiva que devuelva el productorio desde un valor entero hasta otro:

$$prodDesdeHasta\ a\ b\ \Longrightarrow\ a\ *\ (a+1)\ *\ (a+2)\ *\ \ldots\ *\ (b-1)\ *\ b$$

3.11 Escribe una función *variaciones* m n que calcule el número de variaciones de n elementos tomados de m en m. Usa para ello la siguiente relación:

$$variaciones\ m\ n = \frac{m!}{(m-n)!}$$

Escribe otra versión que use esta otra:

$$variaciones\ m\ n = (m-n+1) * (m-n+2) * \ldots (m-1) * m$$

3.12 Escribe una función recursiva que calcule números combinatorios usando la siguiente relación:

$$\left(\begin{array}{c} m \\ n \end{array} \right) = \frac{m!}{(m-n)! \cdot n!}$$

Escribe otra versión que use esta otra:

$$\begin{pmatrix} m \\ 0 \end{pmatrix} = 1$$
$$\begin{pmatrix} m \\ m \end{pmatrix} = 1$$
$$\begin{pmatrix} m \\ n \end{pmatrix} = \begin{pmatrix} m-1 \\ n-1 \end{pmatrix} + \begin{pmatrix} m-1 \\ n \end{pmatrix}$$

¿Puedes garantizar que la última definición acaba?

3.13 Escribe una función que devuelva el i-ésimo número de la sucesión de *fibonacci*. Este sucesión tiene como primer término 0, como segundo 1, y cualquier otro término se obtiene sumado los dos que le preceden: $0, 1, 1, 2, 3, 5, 8, 13, \ldots$

? *fibonacci* 0
0 :: *Integer*

? *fibonacci* 6
8 :: *Integer*

3.14 Escribe una función que determine el mayor de tres números enteros. Escribe otra para cuatro números.

3.15 Escribe una función que tome tres números enteros y devuelva una terna con los números ordenados en orden creciente:

? *ordena*3 10 4 7
$(4, 7, 10)$ :: $(Integer,\ Integer,\ Integer)$

3.16 Escribe una función que determine si un número positivo de exactamente cuatro cifras es capicúa o no:

? *esCapicúa* 1221
True :: *Bool*

? *esCapicúa* 12
ERROR : *número de cifras incorrecto*

3.17 Escribe una función que calcule la suma de las cifras de un número natural:

? *sumaCifras* 123
6 :: *Integer*

3.18 Escribe una función que calcule el número de cifras de un número natural (sin ceros a la izquierda):

? *númeroCifras* 123
3 :: *Integer*

3.19 Define una función

$$aEntero \;::\; [Integer] \rightarrow Integer$$

que transforme una lista de dígitos en el correspondiente valor entero:

? *aEntero* [2, 3, 4]
234 :: *Integer*

Define la función recíproca *aLista*:

? *aLista* 234
[2, 3, 4] :: [*Integer*]

Tema 4

4.1 Consideremos la siguiente función

$$\begin{array}{l} dosVeces \;::\; (Integer \rightarrow Integer) \rightarrow Integer \rightarrow Integer \\ dosVeces\ f\ x \;=\; f\ (f\ x) \end{array}$$

- ¿Cuál es el tipo de la siguiente función?

  $$fun \;=\; dosVeces\ (+1)$$

- Escribe una λ-expresión equivalente a la función *fun*.
- Escribe una sección equivalente a la función *fun*.

4.2 Escribe una función *derivada* que devuelva la derivada de una función de reales en reales usando la aproximación:

$$f'x = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - fx}{\epsilon}$$

Por ejemplo:

$$\begin{array}{l} coseno \;::\; Float \rightarrow Float \\ coseno \;=\; derivada\ sin \end{array}$$

? *derivada sqrt* 1.0
0.499487 :: *Float*

? *coseno* 0.0
1.0 :: *Float*

¿Cuál es el tipo de la función *derivada*?

4.3 Escribe una función *logEnBase* que calcule el logaritmo de un número en una base dada usando la equivalencia:

$$\log_b x = \frac{\ln x}{\ln b}$$

Por ejemplo:

```
log2 :: Float → Float
log2 = logEnBase 2

? logEnBase 2 16
4.0 :: Float

? log2 16
4.0 :: Float
```

¿Cuál es el tipo de la función *logEnBase*?

Tema 5

5.1 ¿Cuáles son los tipos polimórficos de las siguientes funciones?

```
swap (x, y)          = (y, x)
const x y            = x
subst f g x          = f x (g x)
pair (f, g) x        = (f x, g x)
cross (f, g) (x, y)  = (f x, g y)
```

5.2 La función *flip* está predefinida del siguiente modo

```
flip        :: (a → b → c) → b → a → c
flip f x y  = f y x
```

Supongamos la siguiente definición de operador

```
(><) = flip (++)
```

- ¿Cuál es el tipo de (><)?
- Calcula el resultado de la expresión $[1,2,3] \mathbin{><} [5,6]$
- A la vista del ejemplo, ¿para qué crees que sirve la función *flip*?

5.3 ¿Qué hace el siguiente operador?

```
infixr 0 >$>
```

$$
\begin{array}{l}
(>\$>) \quad :: \quad (a \rightarrow a) \rightarrow a \rightarrow a \\
f \;\; >\$> \;\; x \;\; = \;\; f\ (f\ x)
\end{array}
$$

¿por qué su tipo no es $(>\$>) \;\; :: \; (a \rightarrow b) \rightarrow a \rightarrow b$?

5.4 Define el operador $(>\$>)$ del ejercicio anterior usando la composición de funciones (.)

5.5 Consiremos el siguiente operador:

$$
\begin{array}{l}
\textbf{infixl}\ 9 \;\; >.> \\
f \;\; >.> \;\; g \;\; = \;\; \lambda\, x \;\rightarrow\; g\ (f\ x)
\end{array}
$$

- ¿Cuál es su tipo polimórfico?
- ¿Qué hace la siguiente función?

$$
\begin{array}{l}
\mathit{fun} \;\; :: \;\; \mathit{Integer} \rightarrow \mathit{Integer} \\
\mathit{fun} \;\; = \;\; (+2) \;>.>\; (*2) \;>.>\; (+1)
\end{array}
$$

5.6 Consideremos la siguiente función polimórfica

$$
\begin{array}{lcl}
\mathit{iter} & :: & (\mathit{Integer} \rightarrow a \rightarrow a) \rightarrow a \rightarrow \mathit{Integer} \rightarrow a \\
\mathit{iter}\ op\ e\ 0 & = & e \\
\mathit{iter}\ op\ e\ m@(n+1) & = & op\ m\ (\mathit{iter}\ op\ e\ n)
\end{array}
$$

Define usando *iter* funciones tales que

? *listaDecre* 5
$[5,4,3,2,1] \;\; :: \; [\mathit{Integer}]$

? *palos* 3
"III" :: *String*

5.7 La función predefinida *reverse* invierte el orden de los elementos de una lista. Por ejemplo

? *reverse* $[1,2,3]$
$[3,2,1] \;\; :: \; [\mathit{Integer}]$

Defínela y da su tipo polimórfico. ¿Cuál es es tipo de la función $\mathit{palíndromo}\ xs \;\; = \;\; (\mathit{reverse}\ xs == xs)$?

5.8 ¿Cuál es el tipo polimórfico de la función *twice*?

$$\mathit{twice}\ f\ x \;\; = \;\; f\ (f\ x)$$

¿Cuál es el valor de cada una de las siguientes expresiones?

$twice\ (+1)\ 0$
$twice\ twice\ (+1)\ 0$

5.9 La función predefinida *zip* empareja dos listas. Por ejemplo

? $zip\ [1,2,3]\ [4,5,6]$
$[(1,4),\ (2,5),\ (3,6)]\ \ ::\ [(Integer, Integer)]$

? $zip\ [1,2,3]\ [True,\ False]$
$[(1, True),\ (2, False)]\ \ ::\ [(Integer, Bool)]$

? $zip\ [True,\ False]\ [1,2,3]$
$[(True, 1),\ (False, 2)]\ \ ::\ [(Bool, Integer)]$

Defínela y da su tipo polimórfico.

5.10 ¿Cuál es el tipo de las siguientes expresiones (en caso de que sean correctas)?

- $not\ .\ even$
- $even\ .\ not$
- $chr\ .\ ord$
- $ord\ .\ chr$
- $ord\ .\ chr\ .\ (+1)$
- $map\ not$
- $map\ (\lambda\ x\ \rightarrow\ not\ x)$
- $map\ (not\ .\ even)$
- $map\ not\ [True, False]$
- $map\ ord$
- $map\ 1$
- $map\ (+1)$
- $map\ (map\ (+1))$
- $map\ (+\!\!+[1])$
- $map\ (1\ :)$

¿Cuál es el valor de la expresión $map\ (map\ (+1))\ [[1,2,3],\ [10,11]]$?

Tema 6

6.1 Dado el siguiente tipo para representar temperaturas expresadas en dos escalas:

$$\textbf{data}\ \mathit{Temp} \;=\; \mathit{Centígrado}\ \mathit{Float} \mid \mathit{Fahrenheit}\ \mathit{Float}\ \ \textbf{deriving}\ \mathit{Show}$$

escribe una función $\mathit{estáCongelada} :: \mathit{Temp} \to \mathit{Bool}$ que compruebe si una temperatura para el agua está congelada o no.

6.2 Define una función que calcule el perímetro de una *Figura* según están definidas en las transparencias del tema.

6.3 Dadas las siguientes declaraciones de tipo:

$$\begin{array}{lcl}
\textbf{type}\ \mathit{Real} & = & \mathit{Float} \\
\textbf{type}\ \mathit{Imag} & = & \mathit{Float} \\
\textbf{data}\ \mathit{Complejo} & = & \mathit{Real} :- \mathit{Imag}\ \ \textbf{deriving}\ \mathit{Show} \\
\textbf{data}\ \mathit{Resultado} & = & \mathit{UnaReal}\ \mathit{Float} \\
 & \mid & \mathit{DosReales}\ \mathit{Float}\ \mathit{Float} \\
 & \mid & \mathit{DosComplejas}\ \mathit{Complejo}\ \mathit{Complejo} \\
 & & \ \ \textbf{deriving}\ \mathit{Show}
\end{array}$$

completa la función

$$\begin{array}{lcl}
\mathit{raíces} & :: & \mathit{Float} \to \mathit{Float} \to \mathit{Float} \to \mathit{Resultado} \\
\mathit{raíces}\ a\ b\ c & = & \ldots
\end{array}$$

que devuelva las raíces de la ecuación de segundo grado $ax^2 + bx + c = 0$.

6.4 Para el tipo *Nat* definido en las transparencias del tema, define

- Un operador para restar dos naturales
- Un operador para calcular la potencia de naturales
- Una función que convierta un valor de tipo *Integer* en el correspondiente de tipo *Nat*
- Una función que convierta un valor de tipo *Nat* en el correspondiente de tipo *Integer*
- Dos funciones *divNat* y *modNat* que calculen el cociente y resto de dividir dos naturales

6.5 Define las siguientes funciones como una concreciones de *foldNat*:

$$\begin{array}{lcl}
\mathit{suma} & :: & \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat} \\
\mathit{suma}\ m\ \mathit{Cero} & = & m \\
\mathit{suma}\ m\ (\mathit{Suc}\ n) & = & \mathit{Suc}\ (\mathit{suma}\ m\ n)
\end{array}$$

```
prod             :: Nat → Nat → Nat
prod m Cero      =  Cero
prod m (Suc n)   =  suma m (prod m n)
```

Ayuda: Observa que la primera definición,por ejemplo, es equivalente a:

```
suma             :: Nat → Nat → Nat
(suma m) Cero    =  m
(suma m) (Suc n) =  Suc ((suma m) n)
```

6.6 El siguiente tipo puede ser utilizado para representar expresiones aritméticas simples sobre enteros:

```
data Expr = Valor Integer
          | Expr :+: Expr
          | Expr :−: Expr
          | Expr :∗: Expr
            deriving Show
```

Por ejemplo

```
ej1 :: Expr
ej1 =  Valor 5   -- representa el valor 5

ej2 :: Expr
ej2 =  ej1 :+: Valor 3   -- representa el valor 5 + 3

ej3 :: Expr
ej3 =  ej2 :∗: Valor 10   -- representa el valor (5 + 3) * 10
```

Escribe funciones que calculen

- El número de operadores que aparece en una expresión
- El valor de una expresión
- El número de constantes que aparece en una expresión

6.7 Sea el siguiente tipo para representar listas

```
infixl 5 :<
data SnocList a = Vacía | (SnocList a) :< a deriving Show
```

de modo que una lista de la forma $xs :< x$ representa una lista cuyo último elemento es x y el segmento inicial es xs. Por ejemplo

```
l :: SnocList Integer
l =  Vacía :< 1 :< 2 :< 3
```

es una lista cuyo primer elemento es 1, el segundo es 2 y el tercero 3. Para este tipo, define

- una función que devuelva la cabeza (primer elemento) de una lista
- una función que devuelva el último elmento de una lista
- una función que calcule la longitud de una lista
- un operador $+\!+$ para concatenar dos listas
- una función *mapSnocList* análoga a *map* para este tipo de listas

6.8 Sea el siguiente tipo para representar listas

```
infixl 5 :+:
data ConcatList a = V | U a | ConcatList a :+: ConcatList a deriving Show
```

Por ejemplo

```
l :: ConcatList Integer
l = U 1 :+: U 2
```

Para este tipo, define

- una función que devuelva la cabeza (primer elemento) de una lista
- una función que devuelva el último elmento de una lista
- una función que calcule la longitud de una lista
- un operador $+\!+$ para concatenar dos listas
- una función *mapConcatList* análoga a *map* para este tipo de listas

Tema 7

7.1 Sea el siguiente tipo para representar números racionales

```
infix 9 :/
data Racional = Integer :/ Integer
```

Haz este tipo instancia de las clases *Eq*, *Ord*, *Show*, *Num* y *Fractional*.

7.2 Sea el siguiente tipo para representar números naturales

```
data Nat = Cero | Suc Nat deriving Show
```

Haz este tipo instancia de las clases *Eq*, *Ord*, y *Num*.

7.3 La clase predefinida *Functor*

```
class Functor f where
  fmap :: (a → b) → (f a → f b)
```

permite generalizar la función de listas *map* a estructuras polimórficas arbitrarias. Por ejemplo, las instancias predefinidas de esta clase para listas y el tipo *Maybe* son:

```
instance Functor [] where
  fmap = map
instance Functor Maybe where
  fmap f Nothing  = Nothing
  fmap f (Just x) = Just (f x)
```

Realiza instancias para esta clase con los tipos *SnocList* y *ConcatList*.

7.4 Consideremos la siguiente definición del tamaño de un valor:

tamaño x = 1 si x es un número

tamaño x = 1 si x es un carácter

tamaño x = 1 si x es un booleano

tamaño xs = suma de los tamaños de cada elemento de xs si éste es una lista

Define la clase *Medible* que incluya un método para calcular el tamaño de un objeto.
Realiza las instancias correspondientes para la definición anterior.
Por ejemplo

```
? tamaño [[True, False, True], [True, False]]
5 :: Integer
```

7.5 Define la función *ocurrencias* que toma una lista y un dato y devuelve el número de veces que aparece el dato en la lista. ¿Cuál es el tipo de esta función? ¿Cómo puedes utilizar esta función para definir la función *pertenece* que compruebe si un elemento aparece en una lista?

7.6 Define una instancia de *Eq* y otra de *Ord* par el tipo *Color*

```
data Color = Violeta | Azul | Verde | Amarillo | Naranja | Rojo
             deriving Show
```

teniendo en cuenta que dos colores solo son iguales si son idénticos y que el orden de los colores viene dado por la enumeración anterior.

Tema 8

8.1 Define

- una función *partir* que parta una lista en dos mitades.
- una función *mezclar* que tome dos listas ordenadas y devuelve una nueva lista con la mezcla ordenada de ambas.
- usando las funciones anteriores, define la función *mergeSort* que ordene una lista por el método de ordenación por mezcla.

8.2 Define una función *estáOrdenada* que compruebe si una lista está ordenada ascendentemente.

8.3 Define

- una que inserte un dato en su posición adecuada dentro de una lista ordenada.
- usando las función anterior, define la función *insertSort* que ordene una lista por el método de ordenación por inserción.
- expresa la función de ordenación como una concreción de una función de plegado.

8.4 Define

- una función que devuelva los segmentos iniciales de una lista

  ? *inits* $[1,2,3]$
  $[\,[\,],\ [1],\ [1,2],\ [1,2,3]\,]\ \ ::\ [[Integer]]$

- una función que devuelva los segmentos finales de una lista

  ? *tails* $[1,2,3]$
  $[\,[1,2,3],\ [2,3],\ [3],\ [\,]\,]\ \ ::\ [[Integer]]$

- una función que devuelva los segmentos de datos consecutivos de una lista

  ? *segs* $[1,2,3]$
  $[\,[\,],\ [1],\ [2],\ [3],\ [1,2],\ [2,3],\ [1,2,3]\,]\ \ ::\ [[Integer]]$

- una función que devuelva las partes (todas las sublistas) de una lista

  ? *partes* $[1,2,3]$
  $[\,[\,],\ [1],\ [2],\ [3],\ [1,2],\ [2,3],\ [1,3],\ [1,2,3]\,]\ \ ::\ [[Integer]]$

Nota: no es necesario que los resultados aparezcan en el mismo orden que en los ejemplos.

8.5 Da la definición de las funciones predefinidas *head*, *tail*, *last*, *init*, *take*, *drop*, (!!), *takeWhile*, *sum*, *product*, *maximum*, *minimum*, *zip*, *unzip* y *zipWith*.

8.6 Expresa como concreciones de *foldr* y *foldl* las funciones predefinidas *length*, *map f*, *filter p*, $(+\!\!+\ ys)$, *concat*, y *unzip*.

8.7 Alguna de las funciones definidas en la biblioteca *List* son las siguientes:

- *nub* que elimina elementos duplicados de una lista

  ? *nub* $[1,2,3,2,4,2,1]$
  $[1,2,3,4]$ :: $[Integer]$

- *intersperse* que intercala un elemento entre dos consecutivos de una lista

  ? *intersperse* 0 $[1,2,3]$
  $[1,0,2,0,3]$ :: $[Integer]$

- *isPreffixOf* que comprueba si una lista es prefijo de otra

  ? "ab" ‘*isPreffixOf*‘ "abcd"
  True :: *Bool*

- *isSuffixOf* que comprueba si una lista es sufijo de otra
- *delete* que elimina la primera ocurrencia de un dato de una lista

  ? *delete* 1 $[2,1,3,1,1]$
  $[2,3,1,1]$ :: $[Integer]$

- $(\backslash\backslash)$ que realiza la diferencia de listas

  ? $[1,1,2,3,1]$ $\backslash\backslash$ $[1,1,3]$
  $[2,1]$ :: $[Integer]$

Da sus definiciones.

8.8 Usando listas por comprensión, define

- una función *expandir* :: $[Integer] \rightarrow [Integer]$ que reemplace en una lista de números positivos cada número n por n copias de sí mismo:

  ? *expandir* $[1..4]$
  $[1, 2, 2, 3, 3, 3, 4, 4, 4, 4]$ :: $[Integer]$

- la función *concat*

8.9 Mientras que las funciones de plegado pueden ser utilizadas para expresar funciones que consumen listas, las funciones de desplagado se usan para expresar funciones que producen listas. Consideremos las siguiente función de desplegado

```
unfold           :: (b → Bool) → (b → a) → (b → b) → b → [a]
unfold p h t x   =  fun x
  where
    fun x
      | p x        = []
      | otherwise  = h x : fun (t x)
```

que encapsula un patrón de cómputo para producir una lista a partir de cierta semilla x. Si la condición p es cierta para la semilla, entonces se produce la lista vacía (se deja de generar la lista resultado). En otro caso, el resultado es una lista cuya cabeza se obtiene aplicando h a la semilla y cuya cola es una llamada recursiva con semilla $t\ x$. Por ejemplo, la función *desdeHasta* n m que genera la lista $[n..m]$

```
desdeHasta         :: Integer → Integer → [Integer]
desdeHasta n m     =  fun n
  where
    fun n
      | n > m      = []
      | otherwise  = n : fun (n + 1)
```

puede ser definida como concreción de *unfold*:

```
desdeHasta         :: Integer → Integer → [Integer]
desdeHasta n m     =  unfold (> m) id (+1) n
```

- evalúa paso a paso la expresión *desdeHasta* 1 3 usando esta última definición
- define como concreción de *unfold* la función *cuadradosDesdeHasta* n m que devuela $[x^{\wedge} 2 \mid x \leftarrow [n..m]]$
- define como concreción de *unfold* la función *desde* n que devuelva la lista $[n..]$

- define como concreción de *unfold* la función $identidad :: [a] \to [a]$ que devuelva su argumento inalterado
- define como concreción de *unfold* la función *map f*

Tema 9

9.1 Para el tipo *ÁrbolB* definido en las transparencias, define

- una función $todosÁrbolB :: (a \to Bool) \to ÁrbolB\ a \to Bool$ que compruebe si todos los datos almacenados en un árbol cumplen una condición
- una función $algunoÁrbolB :: (a \to Bool) \to ÁrbolB\ a \to Bool$ que compruebe si alguno de los datos almacenados en un árbol cumple una condición

9.2 Consideremos el siguiente tipo para representar árboles binarios con datos de tipo *a* almacenados tan solo en sus hojas

$$\textbf{data}\ ÁrbolH\ a = HojaH\ a \mid NodoH\ (ÁrbolH\ a)\ (ÁrbolH\ a)\ \textbf{deriving}\ Show$$

- define las funciones *tamañoH*, *profundidadH*, *todosÁrbolH* y *algunoÁrbolH* para este tipo de árbol.
- realiza una instancia de la clase *Functor* con este tipo

9.3 Consideremos la siguiente función de plegado para los árboles definidos en el ejercicio previo:

```
foldÁrbolH :: (b → b → b) → (a → b) → (ÁrbolH a → b)
foldÁrbolH f g = fun
  where
    fun (HojaH x)   = g x
    fun (NodoH i d) = f (fun i) (fun d)
```

- da una definición equivalente de *foldÁrbolH* sin **where**.
- define las funciones *tamañoH*, *profundidadH*, *todosÁrbolH* y *algunoÁrbolH* como concreciones de la función *foldÁrbolH*.
- realiza una instancia de la clase *Functor* con este tipo usando *foldÁrbolH*

9.4 Define instancias de *fmap* para árboles binarios y generales usando las funciones de plegado

9.5 Define la función *todosÁrbolB* usando la función de plegado de árboles binarios. Define también una función similar para árboles generales y da definiciones recursivas y con la función de plegado.

9.6 Consideremos la siguiente clase para sobrecargar una función que devuelva el máximo elemento almacenado en una estructura de datos t

```
class TieneMáximo t where
  máximo :: Ord a ⇒ t a → a
```

Realiza instancias de esta clase para listas y los tres tipos de árboles (*ÁrbolB*, *Árbol* y *ÁrbolH*).

9.7 Define una función sobrecargada *ocurrencias* que devuelva cuántas veces aparece un dato en una estructura de datos. Realiza instancias de la clase para listas y los tres tipos de árboles (*ÁrbolB*, *Árbol* y *ÁrbolH*).

9.8 Define una función sobrecargada *hojas* que devuelva los nodos hoja de un árbol (aquellos que no tienen hijos). Realiza instancias de la clase para los tres tipos de árboles (*ÁrbolB*, *Árbol* y *ÁrbolH*).

9.9 Define una función para eliminar un dato de un árbol de búsqueda, de modo el árbol devuelto siga siendo de búsqueda.

Tema 10

10.1 Para la siguiente definición del operador (<+>),

```
(<+>)         :: Nat → Nat → Nat
m <+> Cero    =  m
m <+> Suc n   =  Suc (m <+> n)
```

demostrar que cumple la propiedad asociativa

$$\forall\, x,\ y,\ z :: Nat \,.\, (x \mathbin{<+>} y) \mathbin{<+>} z \equiv x \mathbin{<+>} (y \mathbin{<+>} z)$$

10.2 Para el operador (<+>) del ejercicio anterior

- demuestra que cumple los siguientes lemas

$$\forall\, x :: Nat \,.\, Cero \mathbin{<+>} x \equiv x$$
$$\forall\, x,\ y :: Nat \,.\, Suc\ x \mathbin{<+>} y \equiv Suc\ (x \mathbin{<+>} y)$$

- utilizando los lemas, demuestra que el operador (<+>) es conmutativo

$$\forall\ x,\ y :: Nat\ .\ x\ <+>\ y\ \equiv\ y\ <+>\ x$$

10.3 Demuestra que *map* distribuye sobre la composición (.), usando las definiciones estándar de estas funciones

$$map\ (f.g)\ \equiv\ map\ f\ .\ map\ g$$

es decir,

$$\forall\ xs :: [a],\ g :: a\ \to\ b,\ f :: b\ \to\ c\ .\\ map\ (f\ .\ g)\ xs\ \equiv\ (map\ f\ .\ map\ g)\ xs$$

10.4 Para la definición estándar de $(+\!\!+)$ demuestra que se cumplen las siguientes propiedades

- $(+\!\!+)$ es asociativo

$$\forall\ xs,\ ys,\ zs :: [a]\ .\ (xs\ +\!\!+\ yz)\ +\!\!+\ zs\ \equiv\ xs\ +\!\!+\ (yz\ +\!\!+\ zs)$$

- $[\,]$ es elementro neutro por la derecha

$$\forall\ xs :: [a]\ .\ xs\ +\!\!+\ [\,]\ \equiv\ xs$$

- $[\,]$ es elementro neutro por la izquierda

$$\forall\ xs :: [a]\ .\ [\,]\ +\!\!+\ xs\ \equiv\ xs$$

10.5 Demuestra que las siguientes funciones son equivalentes (calculan lo mismo),

$$\begin{array}{lcl} suma & :: & [Int]\ \to\ Int \\ suma\ [\,] & = & 0 \\ suma\ (x : xs) & = & x\ +\ suma\ xs \end{array}$$

$$\begin{array}{lcl} suma' & :: & [Int]\ \to\ Int \\ suma' & = & foldr\ (+)\ 0 \end{array}$$

es decir, demuestra

$$\forall\ xs :: [Int]\ .\ suma\ xs\ \equiv\ suma'\ xs$$

Nota: Utiliza la defínción estándar de *foldr*:

$$\begin{array}{lcl} foldr\ f\ e\ [\,] & = & e \\ foldr\ f\ e\ (x : xs) & = & f\ x\ (foldr\ f\ e\ xs) \end{array}$$

10.6 Demuestra que *map* no modifica la longitud de una lista:

$$\forall\ xs :: [a],\ f :: a \to b\ .\ length\ (map\ f\ xs) \equiv length\ xs$$

10.7 Demuestra que las dos funciones

$$\begin{array}{lcl} todosÁrbolB & :: & (a \to Bool) \to ÁrbolB\ a \to Bool \\ todosÁrbolB\ p\ VacíoB & = & True \\ todosÁrbolB\ p\ (NodoB\ i\ r\ d) & = & p\ r\ \&\& \\ & & todosÁrbolB\ p\ i\ \&\&\ todosÁrbolB\ p\ d \end{array}$$

$$\begin{array}{lcl} todosÁrbolB' & :: & (a \to Bool) \to ÁrbolB\ a \to Bool \\ todosÁrbolB'\ p & = & foldÁrbolB\ (\lambda\ ti\ r\ td \to p\ r\ \&\&\ ti\ \&\&\ td)\ True \end{array}$$

son equivalentes:

$$\forall\ t :: ÁrbolB\ a,\ \forall\ p :: a \to Bool\ .\ todosÁrbol\ p\ t \equiv todosÁrbol'\ p\ t$$

Nota: Utiliza la defínción de *foldÁrbolB* de las transparencias:

$$\begin{array}{lcl} foldÁrbolB\ f\ z\ VacíoB & = & z \\ foldÁrbolB\ f\ z\ (NodoB\ i\ r\ d) & = & f\ (foldÁrbolB\ f\ z\ i)\ r\ (foldÁrbolB\ f\ z\ d) \end{array}$$

10.8 Enuncia el principio de inducción para el tipo *ÁrbolH*

$$\textbf{data}\ ÁrbolH\ a\ =\ \ HojaH\ a \mid NodoH\ (ÁrbolH\ a)\ (ÁrbolH\ a)$$

Demuestra, mediante dicho principio, que la función

$$\begin{array}{lcl} espejoH & :: & ÁrbolH\ a \to ÁrbolH\ a \\ espejoH\ (HojaH\ x) & = & (HojaH\ x) \\ espejoH\ (NodoH\ i\ d) & = & NodoH\ (espejoH\ d)\ (espejoH\ i) \end{array}$$

verifica la siguiente propiedad:

$$\forall\ t :: ÁrbolH\ a\ .\ espejoH\ (espejoH\ t) \equiv t$$

Tema 11

11.1 Construye redes de procesos y funciones en Haskell para calcular las siguientes listas infinitas

- $[\ False,\ True,\ False,\ True,\ \ldots\ ]$
- $[\ 5,\ 10,\ 15,\ 20,\ \ldots\ ]$
- $[\ 2,\ 4,\ 8,\ 16,\ 32\ \ldots\ ]$
- $[\ [\ 3,4,5\ ],\ [\ 4,5,6\ ],\ [\ 5,6,7\ ],\ \ldots\ ]$

- $[\,[\,1,2,3,\ldots],\ [\,2,3,4,\ldots],\ [\,3,4,5,\ldots],\ \ldots]$
- $[\,[\,1\,],\ [\,2,1\,],\ [\,3,2,1\,],\ \ldots\,]$
- $[\,[\,1\,],\ [\,1,2\,],\ [\,1,2,3\,],\ \ldots\,]$
- $[\,[\,1,2,3\,],\ [\,1,4,9\,],\ [\,1,8,27\,],\ \ldots\,]$
- $[\,1,2,4,7,11,16,22,\ \ldots\,]$

11.2 Construye una red de procesos y la función Haskell para calcular la sucesión cuyos tres primeros términos son 0,1,1 y los demas se obtienen sumando los tres anteriores:

$[0,1,1,2,4,7,13\ \ldots]$

11.3 Se pretende escribir una función *pascal* que devuelva una lista infinita de listas, donde cada lista es una fila del triángulo de *Pascal*:

```
      1
     1 1
    1 2 1
   1 3 3 1
  1 4 6 4 1
 ...........
```

A partir de las siguientes observaciones,

```
  0 1       0 1 1       0 1 2 1
+ 1 0     + 1 1 0     + 1 2 1 0
-----     -------     ---------     ...
  1 1       1 2 1       1 3 3 1
```

construye una red de procesos y la función *pascal* en Haskell. Por ejemplo:

? *take* 5 *pascal*
$[[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1]]$ :: $[[Integer]]$

11.4 Construye redes y funciones que devuelvan listas infinitas con los términos de las siguientes series:

$$expo(x) = 1 + x + \tfrac{1}{2!}x^2 + \tfrac{1}{3!}x^3 + \ldots$$

$$seno(x) = x - \tfrac{1}{3!}x^3 + \tfrac{1}{5!}x^5 - \tfrac{1}{7!}x^7 + \ldots$$

$$coseno(x) = 1 - \tfrac{1}{2!}x^2 + \tfrac{1}{4!}x^4 - \tfrac{1}{6!}x^6 + \ldots$$

Usa las series para calcular aproximaciones a e, $seno(0)$ y $coseno(0)$.