Programación Funcional

Jose Emilio Labra Gayo

Índice General

1 Evolución y Conceptos básicos

‖Falta por hacer: Falta por desarrollar esta sección con más detalle‖

- Bases teóricas: cálculo lambda de Church
- LISP, Funciones de orden superior
- ISWIM, notación infija
- FP, combinadores
- ML, polimorfismo paramétrico, chequeo estático de tipos, inferencia de tipos
- Hope, encaje de patrones
- Miranda, Evaluación perezosa
- Haskell, sobrecarga (clases de tipos), Entrada/Salida (mónadas)

2 Definición de funciones

Los lenguajes funcionales utilizan un modelo computacional simple similar al de una calculadora.

Definición 1 (Función) *Una función entre dos conjuntos* `A` *y* `B` *es una regla que permite asociar a cada elemento* `x` *perteneciente a* `A`*, un elemento* `y` *perteneciente a* `B`

Existen diversos métodos de definición de funciones.

- La notación lambda

  Ejemplo 1 (suma2)

  ```
  > suma3 = \x -> x + 3
  ```

  En Haskell, todas las funciones tienen un único argumento. Para definir funciones de más de un argumento se pueden utilizar dos posibilidades.

  – Mediante tuplas

  Ejemplo 2 (ft)

  ```
  > ft = \(x,y) -> 2 * x + y
  ```

```
?-ft (3,4)
10
```

Mediante curryficación.

– **Definición 2 (currificación)** *La currificación (currying) consiste en simular funciones de varios argumentos mediante funciones de orden superior de un argumento*

Ejemplo 3 (fc) *La función* `fc` *toma un entero* `x` *y devuelve una función que cuando toma un entero* `y` *devuelve* `2*x+y`

```
> fc :: Int -> (Int -> Int)
> fc = \x -> (\y -> 2 * x + y)
```

```
?-fc 3 4
10
```

En realidad, la precedencia de las declaraciones de tipos en Haskell hace innecesarios los paréntesis de la declaración anterior y La notación lambda puede simplificarse mediante encaje de patrones. Dicha definición podría reescribirse como:

Ejemplo 4 (fc')

```
> fc' :: Int -> Int -> Int
> fc' x y = 2 * x + y
```

Las definiciones mediante currificación son muy habituales en Haskell ya que, además de evitar paréntesis innecesarios, facilitan la aplicación parcial de funciones. De esa forma, la expresión `fc 3` tiene significado por sí misma.

```
?-reaplica (fc 3) 4
16
```

En Haskell, una expresión del tipo `(+2)` se denomina una sección y equivale a la aplicación parcial de la función correspondiente, en este caso `(+)`

- Mediante composición de funciones. La composición de funciones es un recurso habitual en la definición de procesos iterativos. La función de composición, aunque está predefinida, podría definirse en Haskell como:

Ejemplo 5 (comp)

```
> comp :: (b -> c) -> (a -> b) -> a -> c
> comp f g = \x -> f ( g x)
```

Ejemplo 6 (fcm)

```
> fcm = comp (+3) (*4)
```

```
?-fcm 2
11
```

La función composición está predefinida como el operador `(.)`.

Ejemplo 7 (fcm') *La función* `fcm'` *equivale a la función* `fcm`

```
> fcm' = (+3) . (*4)
```

Encaje de patrones. Las definiciones de funciones pueden realizarse mediante una serie de ecuaciones. En cada ecuación pueden incluirse varios patrones que el sistema intenta encajar según el orden en que fueron escritos. En cuanto un patrón encaje, se devuelve el valor correspondiente.

- **Ejemplo 8 (fact)** *La función* `fact` *devuelve el factorial de un número.*

```
> fact 0 = 1
> fact n = n * fact (n - 1)
```

```
?-fact 5
120
```

Ecuaciones con guardas. En las definiciones pueden incluirse unas condiciones, denominadas . El sistema evalúa las guardas de cada ecuación. Si encuentra una guarda cuyo valor sea `True`, devuelve el resultado correspondiente.

- **Ejemplo 9 (signo)**

```
> signo x | x > 0     = 1
>  | x == 0    = 0
>  | otherwise = -1
```

La palabra reservada `otherwise` equivale a `True`

- Declaraciones locales. En la definición de funciones es posible utilizar declaraciones locales que facilitan la legibilidad y la reusabilidad del código.

Ejemplo 10 (cuboS) *La función* `cuboS` *calcula el cubo del siguiente de un número*

```
> cuboS x = (x + 1) * (x + 1) * (x + 1)
```

Las declaraciones locales pueden introducirse mediante la combinación `let x = ELocal in Expr`

Ejemplo 11 (cuboS1) `cuboS1` *equivale a* `cuboS` *pero utiliza declaraciones locales.*

```
> cuboS1 x = let sig = x + 1
>          cubo x = x * x * x
>      in cubo sig
```

Otra posibilidad para definir declaraciones locales es la utilización de la palabra `where`

Ejemplo 12 (cuboS2) `cuboS2` *equivale a* `cuboS`

```
> cuboS2 x = cubo sig
>     where sig = x + 1
>    cubo x = x * x * x
```

3 Sistema de Tipos

El universo de posibles valores que puede tomar una expresión se divide en tipos

Definición 3 (Tipo) *Un tipo es un conjunto de valores que puede tomar una expresión.*

En Haskell, un valor pertenece a un único tipo, que es reconocido y comprobado por el sistema sin necesidad de ejecutar el programa. Se realiza, por tanto, una comprobación de tipos en tiempo de compilación. Las principales ventajas son:

- Seguridad, se impide la ejecución de programas que podrían producir errores de tipo.
- Eficiencia, no es necesario realizar comprobaciones de tipo en tiempo de ejecución.

Definición 4 (Sistema de Inferencia de Tipos) *Un sistema de inferencia de tipos permite inferir los tipos de las expresiones sin obligar al programador a su declaración explícita. En caso de que el programador los haya declarado, se comtest que los tipos declarados encajan con los tipos inferidos por el sistema.*

Ejemplo 13 (eligeSaludo)

```
> eligeSaludo x = if x then "Hola, tio"
>                 else "Buenas tardes"
```

No es necesario declarar el tipo de la función `eligeSaludo`*. No obstante, el sistema infiere que tiene el tipo*

```
> eligeSaludo :: Bool -> String
```

La declaración de tipos puede verse como una *especificación* de lo que el programador pretende realizar. Dicha especificación puede comprobarse de forma automática sin necesidad de ejecutar el programa.

3.1 Tipos Predefinidos primitivos

El sistema Haskell cuenta con una serie de tipos primitivos predefinidos. Los más importantes son:

- `Char` indica el conjunto de caracteres con el formato `'a', 'A', '1', '\n', ...`
- `Bool` indica valores booleanos. Pueden ser `True` o `False`
- `Int` indica enteros con límite. Por ejemplo `123, -456, ...`
- `Integer` indica enteros sin límite. Por ejemplo `123123123123, -123456789, ...`
- `Float` indica números en coma flotante. Por ejemplo `12.45, 12e3, ...`
- `()` indica el tipo nulo que solamente incluye el valor `()`

3.2 Tpos definidos por enumeración

El usuario puede definir nuevos tipos de datos mediante la enumeración de sus valores

Ejemplo 14 (Temp)

```
> data Temp     = Calor | Frio | Templado
> data Estacion = Primavera | Otonio | Verano | Invierno
```

Las definiciones de funciones sobre tipos enumerados pueden realizarse mediante encaje de patrones

Ejemplo 15 (tempNormal)

```
> tempNormal :: Estacion -> Temp
> tempNormal    Verano   = Calor
> tempNormal    Invierno = Frio
> tempNormal    _        = Templado
```

Es posible considerar que los tipos predefinidos primitivos han sido definidos originalmente por enumeración.

```
data Bool = True | False
data Char = 'a' | 'b' | 'c' | ...
data Int = -32767 | -32766 | ... | -1 | 0 | 1 | ... 32767
...
```

3.3 Tipos mediante Producto cartesiano

En ocasiones se desea definir un tipo a partir del producto cartesiano de los valores de otros tipos.

Ejemplo 16 (EstadoTemp) *El estado de las temperaturas en una determinada época del año puede definirse como el producto cartesiano de los tipos* `Estacion` *y* `Temp`

```
> data EstadoTemp = ET Estacion Temp
```

Un posible valor sería `ET Verano Templado :: EstadoTemp`

`ET` es un constructor de tipos que actúa como una función `ET :: Estacion -> Temp -> EstadoTemp` que permite obtener un `EstadoTemp` a partir de una estación y una temperatura.

Las funciones sobre los tipos producto también pueden definirse mediante encaje de patrones

Ejemplo 17 (estadoNormal)

```
> estadoNormal :: EstadoTemp -> Bool
> estadoNormal (ET Verano   Calor   ) = True
> estadoNormal (ET Invierno Frio    ) = True
> estadoNormal (ET _        Templado) = True
> estadoNormal (ET _        _       ) = False
```

Ejemplo 18 (Números Complejos) *Los números complejos pueden definirse como el producto de dos números flotantes*

```
> data Complejo = C Float Float
```

```
>      deriving Show
>
> c :: Complejo
> c = C 3 4
```

La expresión `deriving Show` *solicita al sistema Haskell que calcule de forma automática la forma de mostrar valores de tipo* `Complejo`*. De esa forma, será posible evaluar y mostrar valores de tipo* `complejo`*.*

La suma de números complejos podría entonces definirse de la siguiente forma

```
> sumaCom :: Complejo -> Complejo -> Complejo
> sumaCom (C x y) (C x' y') = C (x+x') (y+y')
```

3.4 Combinación de productos y enumeraciones

Es posible definir tipos de datos mediante la combinación de productos cartesianos y enumeraciones (también denominadas sumas o variantes)

Ejemplo 19 (Forma) *El tipo de datos* `Forma` *define figuras geométricas que pueden estar formadas por círculos con un determinado radio o rectángulos con una base y altura determinadas*

```
> data Forma = Cir Float
>            | Rec Float Float
>
> f1, f2 :: Forma
> f1 = Cir 3
> f2 = Rec 4 5
```

Ejemplo 20 (area) *La siguiente función calcula el área de una forma geométrica. Obsérvese que se realiza un encaje de patrones para identificar cuál es la figura geométrica*

```
> area :: Forma  -> Float
```

```
> area (Cir r)   = pi * r * r
> area (Rec a b) = a * b
```

```
 ?-
area f2
```

```
20.0
```

En la definición de un tipo de datos, los constructores de tipo sirven como etiquetas que permiten identificar qué tipo de datos se está utilizando.

Ejemplo 21 (Complejos con representación polar) *En el siguiente ejemplo se define un tipo de datos complejo que admite dos tipos de representaciones: en forma cartesiana y en forma polar. Los constructores de tipo* `Carte` *y* `Polar` *permiten identificar el tipo de representación que se está utilizando.*

```
> data Complejo2 = Carte Float Float
>                | Polar Float Float
>
> c1, c2 :: Complejo2
> c1 = Carte 3 4
> c2 = Polar 4 5
>
```

Ejemplo 22 (mkCarte) *La función* `mkCarte` *convierte un número complejo en el número complejo equivalente representado en forma cartesiana*

```
> mkCarte :: Complejo2 -> Complejo2
> mkCarte (Carte x y ) = Carte x y
> mkCarte (Polar r a ) = Carte (r * cos a) (r * sin a)
```

En las definiciones por encaje de patrones, es posible utilizar el formato `n@p` que asigna el nombre `n` al patrón `p`. De esa forma, la primera línea de la definición anterior podría substituirse por `mkCarte c@(Carte x y) = c`

Ejemplo 23 (sumaC2) *La función* `sumaC2` *suma dos complejos que pueden estar en forma polar o cartesiana. Para realizar la suma, los convierte previamente a forma cartesiana.*

```
> sumaC2 :: Complejo2 -> Complejo2 -> Complejo2
> sumaC2 (Carte x y)   (Carte x' y')  = Carte (x + x') (y + y')
> sumaC2 c@(Carte _ _) p@(Polar _ _)  = sumaC2 c (mkCarte p)
> sumaC2 p@(Polar _ _) c@(Carte _ _)  = sumaC2 (mkCarte p) c
> sumaC2 p@(Polar _ _) p'@(Polar _ _) = sumaC2 (mkCarte p) (mkCarte p')
```

3.5 Registros

En la definición de tipos mediante productos, es posible asignar un nombre a los diferentes campos.

Ejemplo 24 (Persona) *Si se desea definir un tipo de datos que represente personas, puede utilizarse la siguiente declaración*

```
> data Persona = Per { dni :: Integer, nombre :: String, edad :: Int }
>
> pepe = Per { dni = 9987623, nombre = "Jose Ibarra", edad = 28 }
```

Los nombres de los diferentes campos quedan predefinidos como funciones que toman un registro y devuelven el valor correspondiente a ese campo. Por ejemplo `edad :: Persona -> Int`

```
?-edad pepe
28
```

La sintaxis `reg { nombre = valor }` se puede utilizar para generar un nuevo registro igual a `reg` pero con el campo `nombre` igual a `valor`

Ejemplo 25 (incEdad) *La función* `incEdad` *toma una persona y devuelve una nueva persona idéntica a la anterior pero con la edad incrementada*

```
> incEdad :: Persona -> Persona
> incEdad p = p { edad = edad p + 1 }
```

3.6 Tipos parametrizados

El lenguaje Haskell permite definir tipos de datos que toman como parámetro otro tipo de datos

Ejemplo 26 (Par) *El tipo de datos* `Par` *define un tipo de datos formado por pares de elementos del mismo tipo.*

```
> data Par a = P a a
>     deriving Show
>
> p1 :: Par Int
> p1 = P 4 5

> p2 :: Par Char
> p2 = P 'a' 'z'
```

Aunque las tuplas están predefinidas en Haskell. En realidad, podrían definirse como un tipo de datos parametrizado. `data Tupla a b = T a b`

3.7 Tipos recursivos

Es posible definir un tipo de datos en función de sí mismo

Ejemplo 27 (Nat) *El tipo de datos* `Nat` *puede representar números naturales*

```
> data Nat = Cero | Sig Nat
>   deriving Show
>
> n0, n1, n2, n3 :: Nat
> n0 = Cero
> n1 = Sig Cero
> n2 = Sig (Sig Cero)
> n3 = Sig (Sig (Sig Cero))
```

Ejemplo 28 (sumaNat) *La función* `sumaNat` *devuelve la suma de dos números naturales*

```
> sumaNat :: Nat -> Nat -> Nat
> sumaNat Cero y = y
> sumaNat (Sig x) y = Sig (sumaNat x y)
```

```
?-sumaNat n2 n2
Sig (Sig (Sig (Sig Cero)))
```

Ejemplo 29 (prodNat) *La función* `prodNat` *devuelve el producto de dos números naturales*

```
> prodNat :: Nat -> Nat -> Nat
> prodNat Cero y = Cero
> prodNat (Sig x) y = sumaNat (prodNat x y) y
```

```
?-prodNat n2 n3
Sig (Sig (Sig (Sig (Sig (Sig Cero)))))
```

Ejemplo 30 (LInt) *El tipo de datos* `LInt` *representa listas de números enteros*

```
> data LInt = Vacia | Mete Int LInt
>   deriving Show
>
> ls0, ls1, ls2, ls3 :: LInt
> ls0 = Vacia
> ls1 = Mete 4 Vacia
> ls2 = Mete 3 (Mete 4 Vacia)
> ls3 = Mete 6 (Mete 3 (Mete 4 Vacia))
```

Ejemplo 31 (sumaLInt) *La función* `sumaLInt` *devuelve la suma de los elementos de una lista*

```
> sumaLInt :: LInt -> Int
> sumaLInt Vacia = 0
> sumaLInt (Mete n r) = n + sumaLInt r
```

```
?-sumaLInt ls3
13
```

3.8 Tipos recursivos parametrizados

Los tipos recursivos también pueden estar parametrizados

Ejemplo 32 (Lista) *El tipo* `Lista a` *define un tipo de datos recursivo que representa listas de valores de un tipo* `a` *cualquiera.*

```
> data Lista a = VaciaL | MeteL a (Lista a)
>   deriving Show
>
> l1 :: Lista Int
> l1 = MeteL 3 (MeteL 4 (MeteL 2 VaciaL))

> l2 :: Lista Char
> l2 = MeteL 'a' (MeteL 'b' VaciaL)
```

El tipo `Lista a` está predefinido en Haskell como `[a]` y se pueden realizar las siguientes equivalencias

- `VaciaL` se representaría como `[]`
- `MeteL` se representaría mediante el operador infijo `:`
- `MeteL 3 (MeteL 4 (MeteL 2 VaciaL))` se representa como `3 : (4 : (2 : []))`
- La notación `3:(4:(2:[]))` se puede representar como `[3,4,2]`
- En el caso especial de listas de caracteres, `['h','o','l','a']` se puede representar como `"hola"`

Ejemplo 33 (long) *La función* `longLs` *calcula la longitud de una lista*

```
> long :: [a] -> Int
> long [] = 0
> long (x:r) = 1 + long r
```

```
?-long [4,5,6]
3
```

```
?-long "hola"
4
```

Ejemplo 34 (Arbol) *El tipo* `Arbol a` *define un tipo de datos recursivo que representa árboles binarios con valores de tipo* `a`

```
> data Arbol a = Hoja | Rama a (Arbol a) (Arbol a)
>   deriving Show
>
> a1 :: Arbol Int
> a1 = Rama 3 (Rama 4 Hoja Hoja) (Rama 5 (Rama 2 Hoja Hoja) Hoja)
```

Ejemplo 35 (Rosal) *El tipo* `Rosal a` *define un tipo de datos recursivo que representa árboles cuyos nodos pueden tener un número de hijos cualquiera y cuya información está formada por valores de tipo* `a`

```
> data Rosal a = Vacio | Nodo a [Rosal a]
>   deriving Show
>
> r1 :: Rosal Int
> r1 = Nodo 3 [Nodo 4 [], Nodo 5 [Nodo 2 [], Nodo 6 [], Nodo 7 []], Nodo 1 []]
```

4 Técnicas de programación Recursiva

En esta sección se indicarán varios patrones de tratamiento de estructuras recursivas. Estos patrones se aplicarán principalmente a listas, pero muchos de ellos pueden generalizarse a otras estructuras como árboles, rosales, etc.

4.1 Aplicar una función a cada elemento

Ejemplo 36 (aplica)

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

La función `aplica` está predefinida como la función `map`

4.2 Recorrer y transformar una estructura en un valor

Obsérvese la definición de las siguientes funciones

Ejemplo 37 (sumaLs)

```
> sumaLs :: Num n => [n] -> n
> sumaLs [] = 0
> sumaLs (x:xs) = x + sumaLs xs
```

Ejemplo 38 (prodLs)

```
> prodLs :: Num n => [n] -> n
> prodLs [] = 1
> prodLs (x:xs) = x * prodLs xs
```

Ejemplo 39 (longLs)

```
> longLs :: Num n => [a] -> n
> longLs [] = 0
> longLs (x:xs) = 1 + longLs xs
```

En la definición de las siguientes funciones, puede observarse que se utiliza siempre un mismo patrón. Un caso básico cuando la lista está vacía, cuyo valor es `0`, `1` y `0`; y un caso recursivo en el cual se realiza una operación combinando el elemento `x` con el resultado de la llamada recursiva. La operación, en el primer caso es `\x r -> x + r`, en el segundo caso es `\x r -> x * r` y en el último caso es `\x r -> 1 + r`.

El patrón descrito podría generalizarse en una función `foldRight` que tome como parámetros el valor `e` a devolver en el caso básico y la operación `op` a realizar en el caso recursivo.

Ejemplo 40 (foldRight)

```
> foldRight :: (a -> b -> b) -> b -> [a] -> b
> foldRight op e [] = e
> foldRight op e (x:xs) = x `op` foldRight op e xs
```

```
?-foldRight (+) 0 [2,3,4]
9
```

La función `foldRight` está predefinida como la función `foldr`

A partir de la función `foldr`, pueden redefinirse las funciones `sumaLs`,`prodLs` y `longLs`.

Ejemplo 41 (sumaLs')

```
> sumaLs' = foldr (+) 0
```

Ejemplo 42 (prodLs')

```
> prodLs' = foldr (*) 1
```

Ejemplo 43 (longLs')

```
> longLs' = foldr (\x r -> 1 + r) 0
```

Además de las funciones anteriores, existen numerosas funciones que trabajan sobre listas y que pueden definirse a partir de la función `foldr`

Ejemplo 44 (andLs)

```
> andLs :: [Bool] -> Bool
> andLs = foldr (&&) True
```

La función `andLs` está predefinida como la función `and`

Ejemplo 45 (orLs)

```
> orLs :: [Bool] -> Bool
> orLs = foldr (||) False
```

La función `orLs` está predefinida como la función `or`

Ejemplo 46 (append)

```
> append :: [a] -> [a] -> [a]
> append xs ys = foldr (\x r -> x : r) ys xs
```

La función `append` está predefinida como el operador `(++)`

Ejemplo 47 (vuelta)

```
> vuelta :: [a] -> [a]
> vuelta = foldr (\x r -> r ++ [x]) []
```

La función `vuelta` está predefinida como la función `reverse`

Ejemplo 48 (tomaMientras)

```
> tomaMientras :: (a -> Bool) -> [a] -> [a]
> tomaMientras f = foldr (\x r -> if f x then x:r else []) []
```

```
?-tomaMientras (>3) [6,5,2,4,1]
[6,5]
```

La función `tomaMientras` está predefinida como la función `takeWhile`

Ejemplo 49 (filtra)

```
> filtra :: (a -> Bool) -> [a] -> [a]
> filtra f = foldr (\x r -> if f x then x:r else r) []
```

```
?-filtra (>3) [6,5,2,4,1]
[6,5,4]
```

La función `filtra` está predefinida como la función `filter`

4.3 Combinar dos estructuras

Ejemplo 50 (comb)

```
> comb :: [a] -> [b] -> [(a,b)]
> comb xs [] = []
> comb [] ys = []
```

```
> comb (x:xs) (y:ys) = (x,y):comb xs ys
```

```
?-comb [2,3,4] "hola"
[(2,'h'),(3,'o'),(4,'l')]
```

La función `comb` está predefinida como la función `zip`

4.4 Listas intensionales

El lenguaje Haskell admite una notación que facilita la definición de listas. Esta notación es similar a las definiciones matemáticas intensionales

Ejemplo 51 (cpares20) *La función* `cpares20` *calcula los cuadrados de los números pares entre 1 y 20*

```
> cpares20 :: [Integer]
> cpares20 = [x ^ 2 | x <- [1..20], x `mod` 2 == 0]
```

```
?-cpares20
[4,16,36,64,100,144,196,256,324,400]
```

Ejemplo 52 (divisores) *La función* `divisores` *calcula los divisores de un número*

```
> divisores :: Integer -> [Integer]
> divisores n = [d | d <- [1..n], n `mod` d == 0]
```

```
?-divisores 24
[1,2,3,4,6,8,12,24]
```

4.5 Otras posibilidades de recorrer y transformar una lista en un valor

La función `foldr` actúa de la siguiente forma:

```
foldr op e [x1,x2,x3] = x1 `op` (x2 `op` (x3 `op` e))
```

Como puede observarse, la operación se asocia hacia la derecha. El nombre `foldr` viene de fold-right, a la derecha.

Es posible definir una función fold-left que se denominará `foldl` y realiza la asociación a la izquierda.

```
foldl op e [x1,x2,x3] = ((e `op` x1) `op` x2) `op` x3
```

Ejemplo 53 (foldLeft)

```
> foldLeft :: (b -> a -> b) -> b -> [a] -> b
> foldLeft op e [] = e
> foldLeft op e (x:xs) = foldLeft op (e `op` x) xs
```

```
?-foldLeft (+) 0 [1,3,4]
8
```

La función `foldLeft` está predefinida en Haskell como `foldl`

Cuando la operación `op` es asociativa y `e` es el elemento neutro de `op`, se cunple que:

```
foldr op e = foldl op e
```

En general, `foldr` es más eficiente que `foldl`. Sin embargo, `foldl` puede ser útil en cierto tipo de definiciones

Ejemplo 54 (ls2n) *La función* `ls2n` *convierte una lista de dígitos en un número decimal*

```
> ls2n :: [Integer] -> Integer
> ls2n = foldl (\r a -> r * 10 + a) 0
```

```
?-ls2n [1,3,4]
134
```

Si las listas no están vacías, puede ser necesario aplicar la operación sobre los elementos sin utilizar un elemento básico. Los operadores `foldr1` y `foldl1` actúan de la siguiente forma:

```
foldr1 op [x1,x2,x3] = x1 `op` (x2 `op` x3)
foldl1 op [x1,x2,x3] = (x1 `op` x2) `op` x3
```

Ejemplo 55 (maxLs) *La función* `maxLs` *devuelve el valor máximo de una lista de enteros*

```
> maxLs :: [Integer] -> Integer
> maxLs = foldr1 max
```

```
?-maxLs [-1,-3,-4]
-1
```

4.6 Recorrer y transformar una estructura generando resultados parciales

Los operadores predefinidos `scanr`, `scanl`, `scanr1` y `scanl1` recorren la lista generando resultados parciales de la siguiente forma:

```
scanr op e [x1,x2,x3] =
   [x1 `op` (x2 `op` (x3 `op` e)), x2 `op` (x3 `op` e), x3 `op` e, e]
scanl op e [x1,x2,x3] =
   [e, e `op` x1, (e `op` x1) `op` x2, ((e `op` x1) `op` x2) `op` x3]
scanr1 op e [x1,x2,x3] =
   [x1 `op` (x2 `op` x3), x2 `op` x3, x3]
scanl1 op e [x1,x2,x3] =
   [x1, x1 `op` x2, (x1 `op` x2) `op` x3]
```

Ejemplo 56 (ruffini) *La función* `ruffini` *aplica la regla de ruffini*

```
> ruffini :: Float -> [Float] -> [Float]
> ruffini a = scanl1 (\r x -> r * a + x)
```

```
?-ruffini 3 [2,4,3]
[2,10,33]
```

4.7 Generar una estructura

La generación de una lista a partir de un valor puede capturarse mediante la siguiente función

Ejemplo 57 (unfold) *La función* `unfold` *realiza el proceso inverso a la función* `foldr`

```
> unfold :: (a -> Bool) -> (a -> b) -> (a -> a) -> a -> [b]
> unfold p f g x = if p x then []
>                  else f x : unfold p f g (g x)
```

Ejemplo 58 (d2b) *La función* `d2b` *convierte un número decimal en su representación binaria*

```
> d2b :: Integer -> [Integer]
> d2b = reverse . unfold (==0) (`mod` 2) (`div` 2)
```

```
?-d2b 13
[1,1,0,1]
```

Ejemplo 59 (pals) *La función* `pals` *convierte devuelve la lista de palabras que forman una cadena de caracteres*

```
> pals :: String -> [String]
> pals = palsAux . quitaEsp
>   where palsAux  = unfold (=="") tomaPal quitaPal
>         tomaPal  = takeWhile (not . esp)
>         quitaPal = quitaEsp . dropWhile (not . esp)
>         quitaEsp = dropWhile esp
>         esp x    = x == ' ' || x == '\n'
```

```
?-pals "mi perro es listo"
["mi","perro","es","listo"]
```

4.8 Otras estructuras recursivas

Los combinadores definidos en las secciones anteriores pueden generalizarse para otras estructuras recursivas.

Ejemplo 60 (foldA) *La función* `foldA` *recorre y transforma un árbol en un valor*

```
> foldA :: (a -> b -> b -> b) -> b -> Arbol a -> b
> foldA f e Hoja = e
> foldA f e (Rama x i d) = f x (foldA f e i) (foldA f e d)
```

La función `sumaArbol` *calcula la suma de los nodos de un árbol*

```
> sumaArbol :: Num n => Arbol n -> n
> sumaArbol = foldA (\x i d -> x + i + d) 0
```

```
?-sumaArbol a1
14
```

La función `nodosArbol` *devuelve la lista de nodos de un árbol*

```
> nodosArbol :: Arbol a -> [a]
> nodosArbol = foldA (\x i d -> [x] ++ i ++ d) []
```

```
?-nodosArbol a1
[3,4,5,2]
```

Ejemplo 61 (foldRosal) *La función* `foldRosal` *recorre y transforma un rosal en un valor*

```
> foldRosal :: (a -> [b] -> b) -> b -> Rosal a -> b
> foldRosal f e Vacio = e
> foldRosal f e (Nodo x ls) = f x (map (foldRosal f e) ls)
```

La función `sumaRosal` *calcula la suma de los nodos de un rosal*

```
> sumaRosal :: Num n => Rosal n -> n
```

```
> sumaRosal = foldRosal (\x ls -> x + sum ls) 0
```

La función `nodosRosal` *devuelve la lista de nodos de un rosal*

```
> nodosRosal :: Rosal a -> [a]
> nodosRosal = foldRosal (\x ls -> [x] ++ concat ls) []
```

Sería deseable definir una función `fold` genérica que recorriese y transformase todo tipo de estructuras recursivas. Aunque Haskell no permite este tipo de definiciones, existe una extensión del lenguaje denominada *Generic Haskell* que permite definir funciones genéricas, también denominadas politípicas.

4.9 Combinadores recursivos básicos

Muchas estructuras recursivas básicas que en otros lenguajes están predefinidas, pueden ser definidas por el usuario

Ejemplo 62 (hasta) *La función* `hasta` *captura un patrón recursivo básico*

```
> hasta :: (a -> Bool) -> (a -> a) -> a -> a
> hasta p f x = if  p x then x
>               else hasta p f (f x)
```

5 Modelos de Evaluación

5.1 Evaluación impaciente vs. perezosa

En el modelo de evaluación de los lenguajes funcionales, el usuario introduce una expresión que es evaluada por el sistema. La evaluación consiste en buscar subexpresiones dentro de la expresión que puedan transformarse utilizando las funciones predefinidas y las funciones definidas por el usuario. Cuando no es posible realizar más transformaciones se dice que se ha alcanzado la *forma normal*.

Existen dos estrategias fundamentales de evaluación: *evaluación aplicativa* y *evaluación normal*.

En la evaluación aplicativa, se eligen las subexpresiones más internas de la expresión, mientras que en la evaluación normal, se eligen las subexpresiones más externas.

Supóngase que se definen las funciones

```
> cuad x = x * x
> prim (x,y) = x
```

para evaluar `prim (cuad 3,cuad 4)`

- Con evaluación aplicativa, se evalúan primero los argumentos, obteniendo la secuencia

```
  prim (cuad 3, cuad 4)
= prim (3 * 3, cuad 4)
= prim (9, cuad 4)
= prim (9, 4 * 4)
= prim (9,16)
= 9
```

- Con evaluación normal, se evalúan primero las llamadas a la función, obteniendo:

```
  prim (cuad 3, cuad 4)
= cuad 3
= 3 * 3
= 9
```

Como puede observarse, el primer esquema tarda más ya que puede haber argumentos que no se utilizan en el resultado y, sin embargo, son evaluados. Este tipo de modelo también se conoce como *evaluación impaciente* (del inglés *eager*) o estricta. En casos extremos, si alguno de dichos argumentos produce un error (por ejemplo, una división por cero), o no finaliza, el sistema no devolvería el resultado adecuado.

Por ejemplo, al evaluar `prim (cuad 3, 1/0)`

- Con evaluación aplicativa

```
  prim (cuad 3, 1 / 0)
= prim (3 * 3, 1 / 0)
= prim (9, 1 / 0)
= Error, división por cero...
```

- Con evaluación normal, se obtiene

```
  prim (cuad 3, 1 / 0)
= cuad 3
= 3 * 3
= 9
```

La estrategia de evaluación aplicativa también se conoce como *llamada por valor* (*call by value*). Ya que se evalúan primero los argumentos de la función y se le pasan a la función sus valores.

La estrategia de evaluación normal se conoce como *llamada por nombre* (*call by name*), indicando que se pasan las expresiones, en lugar de sus valores. Dichas expresiones no son evaluadas si no se necesita su valor.

En ocasiones, la llamada por nombre puede hacer que ciertas subexpresiones se evalúen más de una vez. Por ejemplo

```
  cuad (5 + 2)
= (5+2) * (5 + 2)
= 7 * (5 + 2)
= 7 * 7
= 49
```

La *evaluación perezosa* permite solucionar este problema mediante una reducción basada en un grafo. Por ejemplo, para la expresión anterior se utilizaría la siguiente secuencia

```
  cuad (5 + 2)
= x * x donde x = 5 + 2
= x * x donde x = 7
= 49
```

5.2 Programación con estructuras infinitas

||Falta por hacer: ||

```
> unos = 1 : unos
> itera f x = x:itera f (f x)
> primos = 1 : map head (itera quitaM [2..])
> quitaM (x:xs) = filtra (\v -> v `mod` x /= 0) xs

> fibo = 1 : 1 : [x + y | (x,y) <- zip fibo (tail fibo)]

> aleas = itera (\x -> (a * x + d) `mod` m)
>    where a = 7
>          d = 0
>          m = 11
```

6 Entrada/Salida

||Falta por hacer: ||

7 Clases de tipos

||Falta por hacer: ||

8 Técnicas de demostración

||Falta por hacer: ||

9 Programación a gran escala

9.1 Sistema de Módulos

||Falta por hacer: ||

9.2 Tipos abstractos de datos

||Falta por hacer: ||

Índice de Materias