Parte II: Programación Funcional

Características Avanzadas: abstracción y orden superior.

P. Julián-Iranzo

Universidad de Castilla-La Mancha

Programación Declarativa

versión Mayo 2014

Índice

1. Objetivo
2. Una reflexión sobre el concepto de abstracción en programación
3. Tipos concretos y tipos abstractos de datos
4. Tipos abstractos de datos y módulos
5. Funciones de orden superior

Índice

1. Objetivo
2. Una reflexión sobre el concepto de abstracción en programación
3. Tipos concretos y tipos abstractos de datos
4. Tipos abstractos de datos y módulos
5. Funciones de orden superior

Objetivo

Este capítulo persigue introducir dos características importantes de los lenguajes de programación funcional modernos que nos permiten escribir programas más robustos y abstractos:

- El uso de tipos abstractos de datos y módulos como una forma de organización del código y ocultación de la información.
- La condición de funciones de orden superior para las funciones, que les otorga el rango de "ciudadanos de primera clase".

Objetivo

Este capítulo persigue introducir dos características importantes de los lenguajes de programación funcional modernos que nos permiten escribir programas más robustos y abstractos:

- El uso de tipos abstractos de datos y módulos como una forma de organización del código y ocultación de la información.
- La condición de funciones de orden superior para las funciones, que les otorga el rango de "ciudadanos de primera clase".

Índice

1. Objetivo
2. **Una reflexión sobre el concepto de abstracción en programación**
3. Tipos concretos y tipos abstractos de datos
4. Tipos abstractos de datos y módulos
5. Funciones de orden superior

Abstracción

- Abstracción es la acción y efecto de abstraer o abstraerse. (Diccionario RAE)
- Abstraer es separar por medio de una operación intelectual las cualidades de un objeto para considerarlas aisladamente o para considerar el mismo objeto en su pura esencia o noción. (Diccionario RAE)
- Es la misma noción de abstracción usada en matemáticas.
- En el contexto de la informática, Hoare describe la abstracción como:

  Abstraction arises from recognition of similarities between certain objects, situations, or processes in the real world, and the decision to concentrate upon those similarities and to ignore for the time being the differences.

Abstracción

- En programación, el término abstracción se asocia con:
  - La estructura de tipos del lenguaje de programación, que determina los recursos para representar y resolver programas: Tipos algebraicos, tipos abstractos de datos y polimorfismo (abstracción de datos).
  - En el diseño orientado a objetos, la abstracción es el medio para la generación de interfaces en los que se disocia la implementación de un objeto de su especificación. En las clases abstractas únicamente se proporciona la estructura de tipos de sus métodos.
  - La idea de evitar la repetición de códio: un fragmento de programa, posiblemente con un parámetro, se dota de un nombre (abstracción procedural). El procedimiento genera un nuevo concepto.
  - La recursión es un ejemplo de mecanismo de abstracción que nos permite ignorar las operaciones y detalles de la máquina subyacente.
  - La idea de resolver un problema en su máxima generalidad (las funciones de orden superior ayudan en esta tarea).

Índice

1. Objetivo
2. Una reflexión sobre el concepto de abstracción en programación
3. **Tipos concretos y tipos abstractos de datos**
4. Tipos abstractos de datos y módulos
5. Funciones de orden superior

Tipos concretos

- Los tipos concretos son aquéllos en los que se describen los valores del tipo y la forma de construirlos, pero no las operaciones sobre esos valores.
- Los tipos algebraicos son un ejemplo de tipos concretos. En Haskell (como hemos visto) se definen mediante una cláusula `data`.
- Los valores de un tipo concreto se nombran mediante un término de datos (formado a partir de los constructores del tipo de datos).
- Se utiliza la técnica de definición por ajuste de patrones para definir (externamente) las operaciones sobre los valores del tipo de datos.

Tipos abstractos de datos

- Un tipo abstracto de datos (TAD) es un conjunto de datos junto con sus operaciones.
- Los tipos abstractos se define especificando y nombrando sus operaciones sobre los valores del tipo en lugar de la forma de construir esos valores.
- Un TAD solamente puede manipularse a través de sus operaciones
- No todas las operaciones son visibles para el usuario del TAD.
- No hay acceso directo a los datos. Por tanto, el usuario no puede utilizar ajuste de patrones para definir funciones sobre el TAD.

Tipos abstractos de datos

- Un TAD se define determinando el nombre y tipo de sus operaciones primitivas así como su especificación.
- La especificación suele hacerse mediante una especificación algebraica o axiomática.

Example (Definición del TAD Cola)

```
--- operaciones primitivas
vacia ::  Cola a
poner ::  a -> Cola a -> Cola a
prim ::  Cola a ->a
elim ::  Cola a -> Cola a
esVacia ::  Cola a -> Bool
```

Example (Definición del TAD Cola (Cont.))

```
--- especificación de las operaciones primitivas
esVacia vacia = True
esVacia (poner x xq) = False
prim (poner x vacia) = x
prim (poner x (poner y xq)) = prim (poner y xq)
elim (poner x vacia) = (vacia)
elim (poner x xq) = (poner x (elim xq))
```

Observación

Suponemos que `poner` *es una función estricta para todo* `x`*, esto es* `poner x ⊥ = ⊥`*. Como consecuencia no se consideran colas parciales ni colas infinitas.*

Tipos abstractos de datos

- Un TAD se define determinando el nombre y tipo de sus operaciones primitivas así como su especificación.
- La especificación suele hacerse mediante una especificación algebraica o axiomática.

Example (Definición del TAD Pila)

```
--- operaciones primitivas
vacia ::  Pila a
esVacia ::  Pila a -> Bool
apilar ::  a -> Pila a -> Pila a
cima ::  Pila a -> a
desapilar ::  Pila a -> (a, Pila a)
```

Tipos abstractos de datos

Example (Definición del TAD Pila (Cont.))

```
--- especificación de las operaciones primitivas
esVacia vacia = True
esVacia (apilar x xp) = False
cima (apilar x xp) = x
desapilar (apilar x xp) = (x, xp)
```

Observación

Las operaciones `cima` *y* `desapilar` *son operaciones parciales. Suponemos que* `cima vacia` $= \perp$ *y* `desapilar vacia` $= \perp$.

Índice

1. Objetivo
2. Una reflexión sobre el concepto de abstracción en programación
3. Tipos concretos y tipos abstractos de datos
4. **Tipos abstractos de datos y módulos**
5. Funciones de orden superior

Módulos

- Haskell proporciona un mecanismo para implementar tipos abstractos de datos: los módulos.
- Las funciones constituyen un medio para dividir un programa en partes.
- Cuando un programa contiene muchas funciones resulta difícil de manejar y conviene estructurarlo en módulos.
- Un módulo define un conjunto de valores, tipos de datos, sinónimos de tipos, clases, y funciones.
- Los módulos permiten dividir un programa en una multiplicidad de ficheros, cada uno de ellos asociados a un módulo.

Módulos

- Tener el código divido en módulos tiene muchas ventajas:
  - Agiliza el desarrollo de programas (cuando varios programadores cooperan).
  - El código se hace más fácil de depurar, actualizar y modificar.
  - Conduce a una aproximación estructurada de la programación.
  - Fomenta la reutilización del código.
- En Haskell, la estructura de un módulo es visible a través de su definición sintáctica.

```
module -> module modid [exports] where body
            | body
body ->  importDecls ; topDecls
            |  importDecls
            |  topDecls
importDecls -> impdecl1 ; ...  ; impdecln (n>=1)
topDecls -> topdecl1 ; ...  ; topdecln (n>=1)
```

Módulos

- Un módulo comienza con una cabecera que contiene una lista de entidades (encerrada entre paréntesis) para ser exportadas.
- La lista de exportación contiene nombres de función o tipos de datos que se desean hacer accesibles desde otros módulos. La sintaxis precisa es:
  - Un valor o método de clase, definido o importado por el módulo.
  - Un tipo de datos algebraico `T`, declarado por una cláusula `data` o `newtype`, que se puede nombrar de tres maneras:
    - `T`;
    - `T(c1, c2, ..., c3)`;
    - `T(..)`

    En el primer caso, el constructor del tipo `T` es accesible, pero los constructores (de datos) de `T` no. Esto facilita la construcción de TADs.
  - Un sinónimo de tipo de datos `T`, declarado por una cláusula `type`
  - Una clase `C` y sus operadores.
  - La forma "`module M`", donde `M` es un módulo.
- La cabecera sigue con una lista (posiblemente vacía) de declaraciones de importación y declaraciones de tipo y de función.

Módulos e implementación de TADs

Example (Implementación del TAD Cola)

```
module Cola(Cola, vacia, esVacia, poner, prim, elim)
where

newtype Cola a = ColaImpl [a]
instance Show a => Show (Cola a) where
         show (ColaImpl xs) = show xs

vacia = ColaImpl []
esVacia (ColaImpl []) = True
esVacia (ColaImpl (_:_)) = False
poner x (ColaImpl xs) = ColaImpl (xs ++ [x])
prim (ColaImpl (x:xs)) = x
elim (ColaImpl (x:xs)) = ColaImpl xs
```

Módulos e implementación de TADs

Example (Implementación del TAD Pila)

```
module Pila (Pila, vacia, esVacia, apilar,
             cima, desapilar) where

newtype Pila a = PilaImpl [a]
instance Show a => Show (Pila a) where
         show (PilaImpl xs) = show xs

vacia = PilaImpl []
esVacia (PilaImpl s) = null s
apilar x (PilaImpl s) = PilaImpl (x:s)
cima (PilaImpl s) = head s
desapilar (PilaImpl (s:ss)) = (s,PilaImpl ss)
```

Módulos y programas

- Un programa Haskell es una colección de módulos, uno de los cuales, por convenio, se llama `Main` y contiene un valor `main` que debe ser exportado.
- El valor del programa es el valor de `main`, que debe ser un cómputo de tipo `IO(t)`, para algún tipo `t`.
- Cuando se ejecuta el programa, se realiza el cómputo `main` y su resultado (de tipo `t`) se desecha.

Example (Estructura de un programa Haskell)

```
module A where
f = ...
```

```
module B where
f = ...
```

```
module Main where
import A
import B
main = A.f >> B.f
```

Módulos y programas

Example (El programa principal)

```
module Main where

import Cola
import Pila

cola ::  Cola Int
cola = (poner 3 (poner 2 (poner 1 Cola.vacia)))

pila ::  Pila Int
pila = (apilar 3 (apilar 2 (apilar 1 Pila.vacia)))

total1 ::  Int
total1 = prim cola

total2 ::  Int
total2 = fst (desapilar pila)

main ::  IO ()
main = print (total1 + total2)
```

Índice

1. Objetivo
2. Una reflexión sobre el concepto de abstracción en programación
3. Tipos concretos y tipos abstractos de datos
4. Tipos abstractos de datos y módulos
5. **Funciones de orden superior**

Derechos civiles y ciudadanos de primera clase

- En Haskell podemos agrupar los distintos objetos sintácticos en: objetos de tipos básidos y definidos por el usuario, listas, tuplas y funciones.
- La filosofía de Haskell es que todos ellos son ciudadanos de primera clase y tienen los mismos "derechos civiles".
- Todos estos objetos pueden:
  - tener nombre;
  - ser los valores de alguna expresión;
  - ser miembros de una lista o elementos de una tupla;
  - pasarse como parámetros de una función o devolverse como resultado de una función.
- Dos clases de objetos que son ciudadanos de segunda, en el mundo de Haskell, son los propios tipos y los módulos.

Funciones como objetos de datos

- De lo anterior se desprende que, una de las características esenciales de los lenguajes de programación funcional es la de tratar las funciones como objetos de datos que pueden ser usadas por otras funciones.
- Hasta el momento hemos puesto énfasis en la definición de funciones y su aplicación a argumentos.
- Si las funciones se tratan como objetos de datos, deberemos permitir también:
  - funciones que tengan por argumentos otras funciones;
  - funciones que produzcan funciones como resultado;
  - estructuras de datos (tales como las tuplas) que contengan funciones como componentes.
- Las funciones con estas características se denominan funciones de orden superior y los lenguajes que las soportan lenguajes de orden superior.

Funciones de orden superior

- En general el tipo de una función de orden superior, f es:

$$\mathsf{f} :: \tau_1 \to \tau_2,$$

Siendo τ_1 o τ_2 tipos que contienen el operador primitivo, $\to$, de construcción de tipos funcionales.

Example (Tipos de funciones de orden superior)

- $$\frac{\tau_1 \text{ es } \mathit{Int} \to \mathit{Int},\ \tau_2 \text{ es } \mathit{Int} \times \mathit{Bool}}{\tau_1 \to \tau_2 \text{ es } (\mathit{Int} \to \mathit{Int}) \to (\mathit{Int} \times \mathit{Bool})}$$

- $$\frac{\tau_1 \text{ es } \mathit{Int} \times \mathit{Bool},\ \tau_2 \text{ es } \mathit{Int} \to \mathit{Int}}{\tau_1 \to \tau_2 \text{ es } (\mathit{Int} \times \mathit{Bool}) \to (\mathit{Int} \to \mathit{Int})}$$

Funciones de orden superior

- Un ejemplo característico de función de orden superior es el operador de composición.
- En Haskell, dicho operador está predefinido y se denota por el símbolo (.). La composición de dos funciones `f` y `g` se define mediante la ecuación: `(f.g)x = f(g x)`
- El operador de composición nos da la oportunidad de definir funciones en un estilo no aplicativo, esto es, un estilo punto a punto:

```
double x = 2*x
treble x = 3*x

sixtimes = double.treble
```

- En lugar de la definición ordinaria:
  `sixtimes x = double (treble x)`

Funciones de orden superior

- El tipo de la función (.) Es complicado pero interesante un sencillo razonamiento nos lleva a deducir el siguiente tipo principal:

```
(.)  ::  (a -> b) -> (c -> a) -> c -> b
```

Observación (Inferencia del tipo de la función (.))

- *El operador (.) se aplica a dos funciones y devuelve otra función. Por tanto, en su forma más general, el tipo que le corresponde es:*

$$(.) :: (\tau_1 \to \tau_2) \to (\tau_3 \to \tau_4) \to \tau_5 \to \tau_6$$

- *Dado que no todas las funciones se pueden componer, se siguen las siguientes restricciones:*
  - $\tau_5 = \tau_3 (= \gamma)$, *el argumento de* `g`;
  - $\tau_6 = \tau_2 (= \beta)$, *el resultado de* `f`;
  - $\tau_4 = \tau_1 (= \alpha)$, *las funciones* `f` *y* `g` *tienen que "encajar".*
- *Lo que nos lleva al tipo principal:* $(\alpha \to \beta) \to (\gamma \to \alpha) \to \gamma \to \beta$

Funciones de Curry

Example

- Supongamos definida la función

  ```
  f ::  Int -> (Int -> Int)
  ```

  entonces tiene sentido escribir `f 5` (una aplicación de `f` sobre `5`)
- Esto constituye una aplicación parcial de la función `f` que da como resultado otra función: `f 5 ::  Int -> Int`
- la aplicación de esta última función sobre `3` también tiene sentido y se escribirá `(f 5) 3`.
- Este es un ejemplo en el que la función `(f 5)` es representada por una expresión compuesta más bien que por un nombre.

- Las funciones que se definen para aceptar argumentos uno a uno, de forma que esto pueden suministrárseles en diferentes puntos de un cómputo o del programa se llaman funciones de Curry.

Funciones de Curry

- Haskell es un lenguaje funcional de orden superior en el que las funciones son explícitamente funciones de curry.

Example

- Cuando definimos la función producto: `cTimes x y = x*y` introducimos la función de orden superior `cTimes::Num a=>a->a->a`
- Tradicionalmente, la función producto se define como una función de dos parámetros: `uTimes(x,y)=x*y` cuyo tipo es `uTimes::Num a=>(a,a)->a`
- Estas funciones parecen formas diferentes de escribir la función producto, pero la primera ofrece algo más. Podemos aplicar parcialmente `cTimes` y obtener una expresión que denota una función.
- Estas expresiones pueden usarse en cualquier contexto en el que una función tenga sentido:

```
treble = cTimes 3        sixTimes = double.treble
```

Funciones de Curry

- Las funciones de Curry son más flexibles que sus contrapartidas no currificadas.
- Cualquier función no currificada que tenga por argumento una tupla

  $$f :: (\alpha_1 \times \alpha_2 \times \ldots \times \alpha_n) \to \beta$$

  puede expresarse como una función de curry.

  $$curryf :: \alpha_1 \to (\alpha_2 \to (\ldots \to (\alpha_n \to \beta)\ldots)$$

- Adoptamos la convención de que las flechas asocian a derechas, con lo que podemos escribir: $curryf :: \alpha_1 \to \alpha_2 \to \ldots \to \alpha_n \to \beta$
- Que se adapta bien con la convención adoptada para la aplicación de funciones (de curry), que asocia a izquierdas. Esto es:

  $$curryf\ E_1\ E_2 \ldots E_n, \text{ donde } E_i :: \alpha_i,$$

  se lee como $(\ldots((curryfE_1)E_2))\ldots E_n)$

Lambda expresiones

- Notación derivada de lambda cálculo de Church (1941), que fue desarrollada para formalizar el concepto de función efectivamente computable antes de que existieran incluso los ordenadores.
- Las lambda expresiones permiten definir funciones anónimas que no tienen asociado un nombre y pueden aplicarse inmediatamente o formar parte de una estructura de datos (como las tupas).
- El recurso de las lambda expresiones será útil cuando una función vaya a usarse de forma inmediata o solamente una vez (e.g. como argumento de una función de orden superior).
- Fijémonos en la definición de una función simple y no recursiva como `doble`:

```
doble x = x + x
```

esta definición liga el nombre doble a un valor y el parámetro `x` sirve como una ayuda en la descripción del valor de la función.

Lambda expresiones

- Haskell posee un modo alternativo de definir la función `doble`:

  ```
  doble = λx -> x + x
  ```

  Aunque esto parece menos legible, muestra el hecho de que `doble` es el nombre que se liga a un valor, que es descrito por la parte derecha de la igualdad y que `x` es parte de la descripción del valor.

- La expresión `λx -> x + x` se denomina lambda expresión (o lambda abstracción) y puede leerse como:

  La función que a `x` *aplica* `x + x`

  proporcionando la descripción de una función anónima, por no tener nombre.

- En una abstracción

  ```
  λx -> E,
  ```

  `x` se denomina variable ligada y `E` se denomina cuerpo. El cuerpo puede ser cualquier expresión sintácticamente legal.

Lambda expresiones

- Las lambda expresiones son funciones de Curry que pueden anidarse

  `λx1 -> λx2 -> ...  -> λxk -> E,`

- La anterior expresión puede escribirse mediante la siguiente notación abreviada:

  `λx1 x2 ...  xk -> E,`

- La sintaxis de una lambda expresión, en su forma más general, es:

  `λp1 p2 ...  pk -> E,`

  donde los `pi` son patrones.

- El conjunto de patrones debe ser lineal (esto es, ninguna variable debe aparecer más de una vez en el conjunto).

Observación

Una expresión como `λx:xs -> x` *es sintacticamente incorrecta; legalmente, debe escribirse como:* `λ(x:xs) -> x`.

Lambda expresiones

Observación

Una lambda expresión como `λx -> E` *no puede ser recursiva, ya que no hay ningún nombre de función asociado a ella.*
Sin embargo, si le asociamos un nombre es posible la recursividad. Como, por ejemplo, definiendo:

```
f = λx -> if x == 0 then 0 else x + f (x-1)
```

- La semántica de una lambda expresión:

  `λp1...pk -> E,`

  puede definirse en términos de la siguiente expresión case equivalente:

  `λx1...xn -> case (x1,...,xn) of (p1,...,pn) -> E`

  donde las `xi` son variables nuevas.

Generalización mediante funciones de orden superior

- Frecuentemente, funciones específicas pueden verse como casos especiales de funciones mucho más generales o como aplicaciones de funciones de orden superior a un argumento particular.

Example

- Dada la función `g :: Int -> Int`. Queremos hallar la función $sumg(n) = \sum_{i=0}^{n} g(i)$, para $n \geq 0$.
- Una definición recursiva en Haskell viene dada por:

```
sumg 0 = (g 0)
sumg n = (sumg (n-1)) + (g n)
```

- El uso de funciones de orden superior supone que podemos tomar `g` como un parámetro explícito de la definición. Entonces podemos definir la función de propósito general `sum`:

```
sum g 0 = (g 0)
sum g n = (sum g (n-1)) + (g n)
```

Patrones de recursión: funciones map, filter y reduce

- Si observamos la definición de las funciones:

```
incList [] = []
incList (x:xs) = (x+1):(incList xs)

makeLists [] = []
makeLists (x:xs) = [x]:(makeLists xs)
```

  Podemos apreciar una misma estructura: ambas realizan una operación simple sobre cada elemento de la lista.

- Decimos que el "patrón de recursión" es el mismo en ambos casos.
- Podemos expresar estos patrones de recursión utilizando funciones de orden superior, que tomen un argumento funcional y lo apliquen a cada uno de los elementos de la lista: Función `map`.

```
map ::  (alpha -> beta) -> [alpha] -> [beta]
map f [] = []
map f (x:xs) = (f x) :  (map f xs)
```

Patrones de recursión: funciones map, filter y reduce

Example

- Funciones auxiliares: `inc x = x + 1` `listify c = [c]`
- Definiciones en términos de la función `map` y argumentos funcionales:

```
incList xs = map inc xs

makeLists xs = map listify xs
```

- Alternativamente, mediante el uso de lamdda expresiones:

```
incList xs = map (λx -> x+1) xs

makeLists xs = map (λc -> [c]) xs
```

Observación

La función `map` tiene el efecto de recrear la lista original, con cada elemento transformado de acuerdo al argumento funcional.

Patrones de recursión: funciones map, filter y reduce

- En ocasiones queremos filtrar ciertos elementos de una lista, quedándonos con los mismos y desechando el resto de los elementos. Por ejemplo:

```
mayoresQue3 [] = []
mayoresQue3 (x:xs) | x > 3 = x:(mayoresQue3 xs)
                   | otherwise = mayoresQue3 xs
```

- Podemos utilizar una función de orden superior, semejante a `map`, que tome como argumento funcional una propiedad `p` y filtre los elementos de la lista deseados (que satisfacen esa propiedad): Función `filter`.

```
filter ::  (alpha -> Bool) -> [alpha] -> [alpha]
filter p [] = []
filter p (x:xs) | (p x) = x:(filter p xs)
                | otherwise = (filter p xs)
```

Patrones de recursión: funciones map, filter y reduce

Example

- Utilizamos la función de orden superior `filter` para redefinir `mayoresQue3`:

```
mayoresQue3 = filter (λx -> x>3)
```

  o bien: `mayoresQue3 = filter (>3)`.

- En ocasiones, deseamos desechar los elementos que satisfacen una propiedad `p`, en vez de conservarlos. Esto también puede hacerse con `filter`:

```
remove p = filter (not.p)
```

- Algunas veces, necesitamos utilizar `filter` y `remove` simultáneamente, para obtener dos lista: una con los elementos que cumplen una propiedad `p` y otra con los que no la cumplen:

```
partition p xs = (filter p xs, remove p xs)
```

Patrones de recursión: funciones map, filter y reduce

- La función `partition` se podría haber definido más eficientemente, evitando recorrer la lista en dos ocasiones:

```
partition p [] = ([],[])
partition p (x:xs)| p x = (x:tieneP, noTieneP)
                  | otherwise = (tieneP,x:noTieneP)
                 where (tieneP,noTieneP) = partition p xs
```

- Ahora, podemos aprovechar la definición de `partition` para implementar el método de ordenación rápido (quicksort):

```
qsort [] = []
qsort (x:xs) = (qsort menores)++(x:qsort mayores)
      where (menores,mayores) = partition (< x) xs
```

Patrones de recursión: funciones map, filter y reduce

- En ocasiones estamos interesados en reducir una lista a un valor.
- Este tipo de operación puede realizarse mediante el empleo de otra función de orden superior:

```
reduce ::  (a -> b -> b) --- Op.  de reducción
                  -> b --- Caso base del resultado
                  -> [a] --- Lista de entrada
                  -> b --- Tipo del resultado
reduce f b [] = b
reduce f b (n:l) = f n (reduce f b l)
```

Observación (Funcionamiento de `reduce`)

La expresión `(reduce f, b, [e1, e2, ..., en])` *es equivalente a la expresión:* `(f e1 (f e2 (... (f en b) ...)))`.
El efecto de la función `reduce` *es sustituir el operador* `(:)` *por la función* `f` *y el operador* `[]` *por el valor base* `b`*, en la lista original.*
El valor base, `b`*, es el valor devuelto al reducir una lista vacía.*

Patrones de recursión: funciones map, filter y reduce

Example (Patrones de recursión con `reduce`)

```
--- Suma de los elementos de una lista de números
suma xs = reduce (+) 0 xs

--- Producto de los elementos de una lista de números
producto xs = reduce (*) 1 xs

--- Máximo de los elementos de una lista de números
maximo xs = reduce max 0 xs
          where max x y | x > y = x
                        | otherwise = y

--- Mínimo de los elementos de una lista de números
minimo [] = 0
minimo (x:xs) = reduce min x xs
              where min x y | x < y = x
                            | otherwise = y
```

Otros patrones de recursión

- Las funciones como `map` y `reduce` abstraen la estructura recursiva común a muchas funciones que procesan listas. Sin embargo, son inapropiadas como funciones que abstraen patrones de recursión para tipos de datos más complejos.
- No hay razón para que no podamos asociar funciones de orden superior semejantes a `map` o `reduce` con tipos más complejos.
- Para concretar vamos a trabajar con la siguiente definición polimórfica del tipo árbol.

```
data Tree a = Nil|Leaf a|Node (Tree a) a (Tree a)
              deriving Show
```

Example (Arbol de tipo `Tree Int`)

```
tree = (Node (Node (Leaf 5) 2 (Leaf 1))
             10
             (Node (Leaf 5) 2 (Leaf 1)))
```

Otros patrones de recursión

Example (Generalización de `map` para árboles `Tree a`)

```
mapTree ::  (a -> b) -> Tree a -> Tree b
mapTree _ Nil = Nil
mapTree f (Leaf n) = Leaf (f n)
mapTree f (Node l n r) =
          Node (mapTree f l) (f n) (mapTree f r)

incTree ::  Num a => Tree a -> Tree a
incTree t = mapTree (+1) t

listTree ::  Tree a -> Tree [a]
listTree t = mapTree (λx -> [x]) t
```

- La función equivalente a `reduce` para árboles es más interesante porque hay varias formas de reducir un árbol.

Otros patrones de recursión

Example (Generalización de `reduce` para árboles `Tree a`)

```
reduceTree ::  (a -> b -> a -> a) -> a -> Tree b -> a
reduceTree _ b Nil = b
reduceTree f b (Leaf n) = f b n b
reduceTree f b (Node l n r) =
            f (reduceTree f b l) n (reduceTree f b r)

--- Suma de los elementos de un árbol
sumTree ::  Num a => Tree a -> a
sumTree t = reduceTree (λl v r -> l+(v+r)) 0 t

--- Aplanamiento de un árbol:  recorrido in-order
flatTree ::  Tree a -> [a]
flatTree t = reduceTree (λl v r -> l ++ (v:r)) [] t

--- Máximo elemento de un árbol
maxElOfTree ::  Tree Integer -> Integer
maxElOfTree = reduceTree (λl v r -> l `max` (v `max` r)) 0
```

Otros patrones de recursión

- Una definición alternativa de `reduceTree` que reduce el subárbol izquierdo usando el resultado de reducir el subárbol derecho como valor base.

Example (Generalización de `reduce` para árboles `Tree a`)

```
reduceTree2 ::  (a -> b -> b) -> b -> Tree a -> b
reduceTree2 _ b Nil = b
reduceTree2 f b (Leaf n) = f n b
reduceTree2 f b (Node l n r) =
              f n (reduceTree2 f (reduceTree2 f b r) l)

--- Aplanamiento de un árbol.  Semejante a flatTree, pero el
--- árbol se recorre en un orden distinto (pre-order).
flatTree2 t = reduceTree2 (:)  [] t
```

Otros patrones de recursión

- Una variación sobre la definición de `reduceTree2`: la llamada a `f` aparece en una posición distinta.

Example (Generalización de `reduce` para árboles `Tree a`)

```
reduceTree3 ::  (a -> b -> b) -> b -> Tree a -> b
reduceTree3 _ b Nil = b
reduceTree3 f b (Leaf n) = f n b
reduceTree3 f b (Node l n r) =
            reduceTree3 f (f n (reduceTree3 f b r)) l

--- Aplanamiento de un árbol.  Idéntico a flatTree,
--- el árbol se recorre en un orden in-order.
--- La ventaja es que el operador (++) no participa
--- en el aplanamiento.
flatTree3 t = reduceTree3 (:)  [] t
```

Otros patrones de recursión

- En `reduceTree3`, el valor base se utiliza como un parámero de acumulación.
- El número de llamadas al operador de construcción de listas `(:)` es $\mathcal{O}(n)$ en lugar de $\mathcal{O}(n^2)$, donde n es el número de elementos en el árbol.
- Existe otras variantes de `reduceTree` que reflejan los distintos modos de recorrer un árbol.

Observación

- *Usando un pequeño conjunto de funciones de orden superior podemos evitar la definición de funciones que 'iteran' sobre elementos de un tipo de datos.*
- *La técnica es comparable a la del polimorfismo de tipos.*
- *Aquí, el uso de funciones de orden superior permite que funciones recursivas, con la misma estructura, sean descritas por una sóla función.*