Tema IV

Programación de Orden Superior

Funciones de orden superior

Son funciones que se aplican sobre otras funciones (funcionales).

Vamos a estudiar unas funciones polimorfas o **esquemas funcionales** que clasificaremos en las siguientes categorías:

- **Filtros**
- **Iteradores**
- **Generadores**
- **Plegadores** o transformadores de estructuras

Filtros (I)

Funcionales para reducir o filtrar una estructura con la ayuda de un predicado. Dentro de esta categoría tenemos:

Funcional para eliminar los elementos de una lista que no cumplen un predicado:

```
filter :: (a->Bool)->[a]->[a]
filter _ [] = []
filter p (x:xs) | p x = x:xs'
                | otherwise = xs'
                  where xs'=filter p xs
```

Filtros (II)

Funcional para producir el segmento inicial de mayor longitud de una lista cuyos elementos cumplen un cierto predicado:

```
takeWhile :: (a->Bool)->[a]->[a]
takeWhile _ [] = []
takeWhile p (x:xs) | p x = x:takeWhile p xs
                   | otherwise = []
```

Funcional para producir el segmento complementario:

```
dropWhile :: (a->Bool)->[a]->[a]
dropWhile _ [] = []
dropWhile p (x:xs) | p x = dropWhile p xs
                   | otherwise = x:xs
```

Filtros (III)

Funcional para producir el segmento con todos los elementos de una lista hasta el primero que cumple un cierto predicado:

```
takeUntil :: (a->Bool)->[a]->[a]
takeUntil _ [] = []
takeUntil p (x:xs) |p x = [x]
                   |otherwise = x : takeUntil p xs
```

Aplicaciones

```
mayores :: Int->[Int]->[Int]
mayores x = filter (x<)

eliminar ::Eq a => a->[a]->[a]
eliminar x = filter (x/=)
```

Algunas propiedades

```
p,q :: a->Bool .
(filter p · filter q = filter q · filter p)

xs, ys :: [a], p :: a->Bool .
(filter p (xs++ys) = filter p xs ++ filter p ys)

xs :: [a], p :: a->Bool .
(takeWhile p xs ++ dropWhile p xs = xs)
```

Ejercicios

Definid los funcionales siguientes:

- **`test`**, que compruebe si todos los elementos de una lista cumplen un determinado predicado.
- **`primero`**, que calcule el primer elemento de una lista que cumple un cierto predicado.
- **`separar`**, que separe una lista en un par de listas, una con todos los elementos que cumplen un cierto predicado y la otra con los elementos que no lo cumplen.

Iteradores (I)

Funcionales para la aplicación reiterada de una función. Dentro de esta categoría tenemos:

Funcional para la aplicación de una función a cada uno de los elementos de una lista:

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

Funcional para la aplicación de una función a cada uno de los elementos de un árbol de hojas:

```
mapHTree :: (a->b)->ArbolH a->ArbolH b
mapHTree f (H x) = H (f x)
mapHTree f (F x y) = F (mapHTree x) (mapHTree y)
```

Iteradores (II)

Funcional para la aplicación reiterada de una función a un valor hasta alcanzar un resultado que cumpla una condición:

```
until :: (a->Bool)->(a->a)->a->a
until p f x | p x = x
            | otherwise = until p f (f x)
```

(Esquema correspondiente a un ciclo **`While not p Do f End`**)

Iteradores (III)

Funcionales para generar listas por aplicación reiterada de una función a un valor dado:

- Lista infinita

```
iterate :: (a->a)->a->[a]
iterate f x = x : iterate f (f x)
```

- Lista hasta que se cumpla una condición

```
iterateUntil :: (a->Bool)->(a->a)->a->[a]
iterateUntil p f x
           | p x = x : []
           | otherwise = x : listUntil p f (f x)
```

Ejercicios

Utilizando los funcionales anteriores, definid

- Una expresión para la lista de las potencias de 2.
- Un funcional para generar la lista infinita siguiente
  [f x, x, f (f (f x)), f (f x), f (f (f (f (f x)))), f (f (f (f x))), ...
- Una función para calcular la suma de los cuadrados de los n primeros números naturales.

Notaciones especiales para generadores de listas

- `[n..]`

  ```
  iterate (+1) n
  ```

- `[n,m..]`

  ```
  iterate (+(m-n)) n
  ```

- `[n..p]`

  ```
  takeWhile (<= p)(iterate (+1) n)
  ```

- `[n,m..p]`

  ```
  takeWhile ((if m>=n then (>=) else (<=)) p)
                  iterate (+(m-n))
  ```

Notación ZF (Zermelo-Fraenkel) para listas

- `[expresión | calificador {, calificador}]`
- `calificador:`
  - generador `pat <- exp. de lista`
  - guarda `exp. booleana`
  - def. local `let pat = exp`
- `R. del generador:`

```
[e|pat<-xs, Q] = concat (map pt xs)
                 where pt x = case x of
                                pat -> [e|Q]
                                 _  -> []
```

- `R. de la guarda:`

```
[e|p, Q] = if p then [e|Q] else []
```

- `R. de la def. local:`

```
[e|let pat=exp, Q] = [e|Q] where pat=exp
```

Notación ZF: Ejemplos (I)

```
[f x| x<-xs]           = map f xs
[f x| x<-xs, p x]      = map f (filter p xs)
[f x y|x<-xs,y<-ys] =
                    concat [[f x y|y<-ys]|x<-xs]
[2| even 4]         = [4]
[2| 2>3]            = []
[x| x+3<-[1..4]] = [0,1]
[y| (3,y)<-[(3,2),(5,1),(3,5)]] = [2,5]
[5| x+3<-[1..4]] = [5,5]
[x*x| x<-[1..10],even x] = [4,16,36,64,100]
[x*x| x+3<-[1..8],even (x+1)] = [1,9,25]
```

Notación ZF: Ejemplos (II)

- `[(x,y)| x<-[1..2],y<-[1..2]] =`
  `[(1,1),(1,2),(2,1),(2,2)]`
- `[x| x<-[1..3],y<-[1..2]] = [1,1,2,2,3,3]`
- `[3*x| let x+2 = 4]          = [6]`
- Ternas de números naturales que cumplen el teorema de Pitágoras (ternas pitagóricas):

```
t_pit n = [(x,y,z)|x<-ns,y<-ns,z<-ns,
               x*x+y*y==z*z] where ns = [1..n]
```

- Lista de los números primos:

```
criba (p:xs) = [x|x<-xs, x`mod`p /= 0]
listaPrimos = map head (iterate criba [2..])
```

Notación ZF: Observaciones

- Los calificadores pueden utilizar valores generados por calificadores anteriores:

```
[(x,y)|x<-[1..3],y<-[x+1..4]] =
     [(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]
```

- Las variables de los calificadores posteriores se imponen sobre las variables de los calificadores anteriores

```
[x|x<-[1,2],x<-[3,4]] = [3,4,3,4]

[x|x<-[1..3],let x=5] = [5,5,5]
```

- El orden de las guardas y las definiciones locales influyen en la eficiencia de los cálculos

```
[(x,y)|x<-[1..3],even x,y<-[1..2]]
[facX + y|x<-[1..3],let facX = fact x, y<-[1,2]]
```

Plegadores

- Son funciones de orden superior que sintetizan los esquemas recursivos que se utilizan con las distintas estructuras recursivas.
- Para cada estructura (lista, árbol,...) se puede definir un plegador adecuado.
- Para algunas estructuras (listas) con las que se pueden utilizar varios esquemas recursivos (r. lineal, r. de cola), se pueden definir otros tantos plegadores.

Recursión sobre listas

Recursión lineal:

```
suma [] = 0
suma (x:xs) = x + suma xs
```

Recursión de cola:

```
suma_ac a [] = a
suma_ac a (x:xs) = suma_ac (a+x) xs
```

Plegadores (I)
Esquemas recursivos sobre listas: Recursión lineal simple

- Definición por recursión lineal:

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

- Elementos que caracterizan la definición:

```
f :: a->b->b
z :: b
```

- Esquema funcional:

```
foldr :: (a->b->b)->b->[a]->b
foldr _ z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

```
g = foldr f z
```

Recursión lineal. Ejemplos (I)

- ```
  suma [] = 0
  suma (x:xs) = x + suma xs
  ```
  `suma = foldr (+) 0`
- ```
  length [] = 0
  length (x:xs) = 1 + length xs
                = f x (length xs)
                  where f x y = 1 + y
  ```
  `length = foldr (\x y -> 1+y) 0`
- ```
  inversa [] = []
  inversa (x:xs) = inversa xs ++ [x]
                 = f x (inversa xs)
                   where f x y = y ++ [x]
  ```
  `inversa = foldr (\x y -> y++[x]) []`
  ```

Recursión lineal. Ejemplos (II)

- ```
  esta _ [] = False
  esta y (x:xs) = y==x || esta y xs
                = f x (esta y xs)
                  where f x r = y==x || r
  esta y = foldr (\x r -> y==x || r) False
  ```
- ```
  map g [] = []
  map g (x:xs) = g x : map g xs
               = f x (map g xs)
                 where f x r = g x : r
  map g = foldr (\x r -> g x : r) []
  ```

Plegadores (II)
Esquemas recursivos sobre listas: Recursión de cola con acumulador

- Definición por recursión de cola (con acumulador):

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

- Elementos que caracterizan la definición:

```
f :: b->a->b
```

- Esquema funcional:

```
foldl :: (b->a->b)->b->[a]->b
foldl _ z [] = z
foldl f z (x:xs) = foldl f (f z x) xs
```

```
g  = foldl f
```

Recursión de cola. Ejemplos (I)

- ```
  suma_ac a [] = a
  suma_ac a (x:xs) = suma_ac (a+x) xs
  ```
  `suma_ac = foldl (+)`

- ```
  inversa_ac a [] = a
  inversa_ac a (x:xs) = inversa_ac (x:a) xs
                      = inversa_ac (f a x) xs
                        where f a x = x:a
  ```
  `inversa_ac = foldl (\x y -> y:x)`

Recursión sobre árboles

Árboles de hojas:

```
n_hojas (H x) = 1
n_hojas (B iz de) = n_hojas iz + n_hojas de
```

Árboles homogéneos:

```
postorden Vacio = []
postorden (Nodo r i d) = postorden i ++
                           postorden d ++ [r]
```

Árboles generales:

```
postorden (Arbol r []) = [r]
postorden (Arbol r (t:ts)) =
          postorden t ++ postorden (Arbol r ts)
```

(Plegadores III) Esquema recursivo sobre **árboles de hojas**

- Las funciones definidas sobre árboles del tipo **`ArbolH a`** frecuentemente presentan el siguiente esquema recursivo:

```
g :: ArbolH a -> b
g (H x)       = h x
g (B iz de) = f (g iz) (g de)
```

- Elementos característicos de la definición:

`f::b->b->b    h::a->b`

- Esquema funcional adaptable a cada definición:

```
foldH :: (b->b->b) -> (a->b) -> ArbolH a -> b
foldH _ h (H x)      = h x
foldH f h (B iz de)= f (foldH f h iz) (foldH f h de)
```

Recursión sobre árboles de hojas. Ejemplos

- ```
  n_hojas (H x) = 1
  n_hojas (B iz de) = n_hojas iz + n_hojas de
  n_hojas = foldH (+) (\x->1)
  ```

- ```
  alturaH (H x) = 0
  alturaH (B iz de)
        = 1 + max (alturaH iz) (alturaH de)
        = f (alturaH iz) (alturaH de)
          where f x y = 1 + max x y
  alturaH = foldH f (\x->0)
             where f x y = 1 + max x y
  ```

(Plegadores IV) Esquema recursivo sobre **árboles de homogéneos**

- Las funciones definidas sobre árboles del tipo **`ArbolB a`** frecuentemente presentan el siguiente esquema recursivo:

```
g :: ArbolB a -> b
g Vacio           = z
g (Nodo r iz de) = f r (g iz) (g de)
```

- Elementos característicos de la definición:

`f::a->b->b->b`   `z::b`

- Esquema funcional adaptable a cada definición:

```
foldB :: (a->b->b->b) -> b -> ArbolB a -> b
foldB _ z Vacio           = z
foldB f z (Nodo r iz de) = f r (foldB f z iz) (foldB f z de)
```

Recursión sobre árboles homogéneos. Ejemplos

```
n_nodos Vacio = 0
n_nodos (Nodo r iz de) =
      1 + n_nodos iz + n_nodos de
n_nodos = foldB (\r i d -> 1+i+d) 0
```

```
alturaB Vacio = 0
alturaB (Nodo r iz de)
      = 1 + max (alturaB iz) (alturaB de)
      = f r (alturaB iz) (alturaB de)
        where f r x y = 1 + max x y
alturaB = foldB f 0
        where f r x y = 1 + max x y
```

(Plegadores V) Esquema recursivo sobre **árboles generales**

- Las funciones definidas sobre árboles del tipo **`ArbolG a`** frecuentemente presentan el siguiente esquema recursivo:

```
g :: ArbolG a -> b
g (Arbol r [])   = h x
g (Arbol r (t:ts)) = f (g t) (g (Arbol r ts))
```

- Elementos característicos de la definición:

  `f::b->b->b    h::a->b`

- Esquema funcional adaptable a cada definición:

```
foldG :: (b->b->b) -> (a->b) -> ArbolG a -> b
foldG _ h (Arbol r [])     = h x
foldG f h (Arbol r (t:ts))= f (foldG f h t)
                              (foldG f h (Arbol r ts))
```

Recursión sobre árboles generales. Ejemplos

- ```
  n_nodosG (Arbol r []) = 1
  n_nodosG (Arbol r (t:ts) =
       n_nodosG t + n_nodosG (Arbol r ts)
  ```
  `n_nodosG = foldG (+) (\x->1)`

- ```
  alturaG (Arbol r []) = 1
  alturaG (Arbol r (t:ts)
  = max (1 + alturaG t) (alturaG (Arbol r ts))
  = f (alturaG t) (alturaG (Arbol r ts))
        where f x y = max (1+x) y
  ```
  `alturaG = foldG f (\x->1)`
  `           where f x y = max (1+x) y`