Programación declarativa (2007–08)

Tema 4: Aplicaciones de programación funcional

José A. Alonso Jiménez

Grupo de Lógica Computacional
Departamento de Ciencias de la Computación e I.A.
Universidad de Sevilla

Tema 4: Aplicaciones de programación funcional

1. Funciones combinatorias
   - Segmentos y sublistas
   - Permutaciones y combinaciones
   - El patrón alias `@`
2. Rompecabezas lógicos
   - El problema de las reinas
   - Números de Hamming
3. Búsqueda en grafos
   - Búsqueda en profundidad grafos

Tema 4: Aplicaciones de programación funcional

Segmentos iniciales

- `iniciales l` es la lista de los segmentos iniciales de la lista `l`.
  Por ejemplo,

```
iniciales [2,3,4]   ⇝ [[],[2],[2,3],[2,3,4]]
iniciales [1,2,3,4] ⇝ [[],[1],[1,2],[1,2,3],[1,2,3,4]]
```

```
iniciales :: [a] -> [[a]]
iniciales []     = [[]]
iniciales (x:xs) = [] : [x:ys | ys <- iniciales xs]
```

Segmentos finales

- `finales l` es la lista de los segmentos finales de la lista `l`. Por ejemplo,

```
finales [2,3,4]   ⇝ [[2,3,4],[3,4],[4],[]]
finales [1,2,3,4] ⇝ [[1,2,3,4],[2,3,4],[3,4],[4],[]]
```

```
finales :: [a] -> [[a]]
finales []     = [[]]
finales (x:xs) = (x:xs) : finales xs
```

Segmentos

- `segmentos l` es la lista de los segmentos de la lista `l`. Por ejemplo,

```
Main> segmentos [2,3,4]
[[],[4],[3],[3,4],[2],[2,3],[2,3,4]]
Main> segmentos [1,2,3,4]
[[],[4],[3],[3,4],[2],[2,3],[2,3,4],[1],[1,2],[1,2,3],[1,
```

```
segmentos :: [a] -> [[a]]
segmentos []     = [[]]
segmentos (x:xs) =
    segmentos xs ++ [x:ys | ys <- iniciales xs]
```

Sublistas

- `sublistas l` es la lista de las sublistas de la lista `l`. Por ejemplo,

```
Main> sublistas [2,3,4]
[[2,3,4],[2,3],[2,4],[2],[3,4],[3],[4],[]]
Main> sublistas [1,2,3,4]
[[1,2,3,4],[1,2,3],[1,2,4],[1,2],[1,3,4],[1,3],[1,4],[1],
 [2,3,4],   [2,3],  [2,4],  [2],  [3,4],  [3],  [4], []]
```

```
sublistas :: [a] -> [[a]]
sublistas []     = [[]]
sublistas (x:xs) = [x:ys | ys <- sub] ++ sub
                   where sub = sublistas xs
```

Tema 4: Aplicaciones de programación funcional

Permutaciones

- `permutaciones l` es la lista de las permutaciones de la lista `l`. Por ejemplo,

```
Main> permutaciones [2,3]
[[2,3],[3,2]]
Main> permutaciones [1,2,3]
[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
```

- Definición recursiva:

```
permutaciones :: Eq a => [a] -> [[a]]
permutaciones [] = [[]]
permutaciones xs =
    [a:p | a <- xs, p <- permutaciones(xs \\ [a])]
```

- `xs \\ ys` es la lista de los elementos de `xs` que no pertenecen a `ys`. Para usarla, hay que importarla: `import Data.List ((\\))`

Permutaciones

- Definición alternativa:

```
permutaciones' :: [a] -> [[a]]
permutaciones' []     = [[]]
permutaciones' (x:xs) = [zs | ys <- permutaciones' xs,
                              zs <- intercala x ys]
```

donde `intercala x l` es la lista de las listas obtenidas intercalando `x` entre los elementos de la lista `l`. Por ejemplo,
`intercala 1 [2,3]` $\rightsquigarrow$ `[[1,2,3],[2,1,3],[2,3,1]]`

```
intercala :: a -> [a] -> [[a]]
intercala e []     = [[e]]
intercala e (x:xs) =
    (e:x:xs) : [(x:ys) | ys <- (intercala e xs)]
```

Permutaciones

- La segunda definición es más eficiente. En efecto,

```
Main> :set +s
Main> permutaciones [1,2,3]
[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
(429 reductions, 812 cells)
Main> permutaciones' [1,2,3]
[[1,2,3],[2,1,3],[2,3,1],[1,3,2],[3,1,2],[3,2,1]]
(267 reductions, 485 cells)
```

Combinaciones

- `combinaciones n l` es la lista de las combinaciones `n`–arias de la lista `l`. Por ejemplo,

  ```
  combinaciones 2 [1,2,3,4] ⇝ [[1,2],[1,3],[1,4],[2,3],[2,
  ```

- Definición mediante `sublistas`:

```
combinaciones :: Int -> [a] -> [[a]]
combinaciones n xs =
    [ys | ys <- sublistas xs, length ys == n]
```

Combinaciones

- Definición directa:

```
combinaciones' :: Int -> [a] -> [[a]]
combinaciones' 0 _          = [[]]
combinaciones' _ []         = []
combinaciones' (n+1) (x:xs) =
    [x:ys | ys <- combinaciones' n xs] ++
    combinaciones' (n+1) xs
```

- Comparación de eficiencia:

```
Main> combinaciones 1 [1..10]
[[1],[2],[3],[4],[5],[6],[7],[8],[9],[10]]
(42363 reductions, 52805 cells)
Main> combinaciones' 1 [1..10]
[[1],[2],[3],[4],[5],[6],[7],[8],[9],[10]]
(775 reductions, 1143 cells)
```

Tema 4: Aplicaciones de programación funcional

El patrón alias @

- Definición de `finales` con patrones:

```
finales_1 []     = [[]]
finales_1 (x:xs) = (x:xs) : finales_1 xs
```

- Definición de `finales` con selectores:

```
finales_2 [] = [[]]
finales_2 xs = xs : finales_2 (tail xs)
```

- Definición de `finales` con alias::

```
finales_3 []       = [[]]
finales_3 l@(x:xs) = l : finales_3 xs
```

- El patrón `l@(x:xs)` se lee "`l` como `x:xs`".

Definición con alias

- `dropWhile p l` es la lista `l` sin los elementos iniciales que verifican el predicado `p`. Por ejemplo,

```
dropWhile even [2,4,6,7,8,9] ⇝ [7,8,9]
```

Prelude

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

Tema 4: Aplicaciones de programación funcional

El problema de las N reinas

- Enunciado: Colocar N reinas en un tablero rectangular de dimensiones N por N de forma que no se encuentren más de una en la misma línea: horizontal, vertical o diagonal.
- El problema se representa en el módulo `Reinas`. Importa la diferencia de conjuntos `(\\)` del módulo `List`:

```
module Reinas where
import Data.List ((\\))
```

- El tablero se representa por una lista de números que indican las filas donde se han colocado las reinas. Por ejemplo, `[3,5]` indica que se han colocado las reinas `(1,3)` y `(2,5)`.

```
type Tablero = [Int]
```

El problema de las N reinas

- `reinas n` es la lista de soluciones del problema de las N reinas. Por ejemplo, `reinas 4` $\rightsquigarrow$ `[[3,1,4,2],[2,4,1,3]]`. La primera solución `[3,1,4,2]` se interpreta como

	R		
			R
R			
		R	

```
reinas :: Int -> [Tablero]
reinas n = aux n
    where aux 0     = [[]]
          aux (m+1) = [r:rs | rs <- aux m,
                              r <- ([1..n] \\ rs),
                              noAtaca r rs 1]
```

El problema de las N reinas

- `noAtaca r rs d` se verifica si la reina `r` no ataca a niguna de las de la lista `rs` donde la primera de la lista está a una distancia horizontal `d`.

```
noAtaca :: Int -> Tablero -> Int -> Bool
noAtaca _ [] _ = True
noAtaca r (a:rs) distH = abs(r-a) /= distH &&
                         noAtaca r rs (distH+1)
```

- Esta solución está basada en [**?**] p. 95.

Tema 4: Aplicaciones de programación funcional

Números de Hamming

- Enunciado: Los números de Hamming forman una sucesión estrictamente creciente de números que cumplen las siguientes condiciones:
  1. El número 1 está en la sucesión.
  2. Si x está en la sucesión, entonces $2x$, $3x$ y $5x$ también están.
  3. Ningún otro número está en la sucesión.
- `hamming` es la sucesión de Hamming. Por ejemplo,

```
take 12 hamming ⇝ [1,2,3,4,5,6,8,9,10,12,15,16]
```

```
hamming :: [Int]
hamming = 1 : mezcla3 [2*i | i <- hamming]
                      [3*i | i <- hamming]
                      [5*i | i <- hamming]
```

Números de Hamming

- `mezcla3 xs ys zs` es la lista obtenida mezclando las listas ordenadas `xs`, `ys` y `zs` y eliminando los elementos duplicados. Por ejemplo,

```
Main> mezcla3 [2,4,6,8,10] [3,6,9,12] [5,10]
[2,3,4,5,6,8,9,10,12]
```

```
mezcla3 :: [Int] -> [Int] -> [Int] -> [Int]
mezcla3 xs ys zs = mezcla2 xs (mezcla2 ys zs)
```

Números de Hamming

- `mezcla2 xs ys zs` es la lista obtenida mezclando las listas ordenadas `xs` e `ys` y eliminando los elementos duplicados. Por ejemplo,

```
Main> mezcla2 [2,4,6,8,10,12] [3,6,9,12]
[2,3,4,6,8,9,10,12]
```

```
mezcla2 :: [Int] -> [Int] -> [Int]
mezcla2 p@(x:xs) q@(y:ys) | x < y     = x:mezcla2 xs q
                          | x > y     = y:mezcla2 p  ys
                          | otherwise = x:mezcla2 xs ys
mezcla2 []        ys                  = ys
mezcla2 xs        []                  = xs
```

Tema 4: Aplicaciones de programación funcional

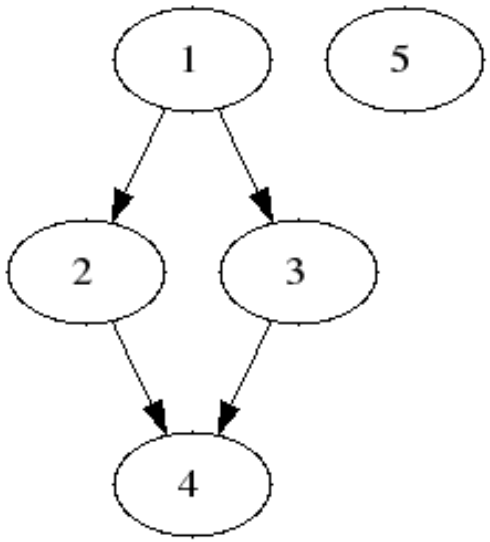
Representación de grafos

- Un grafo de tipo `v` es un tipo de datos compuesto por la lista de vértices y la función que asigna a cada vértice la lista de sus sucesores.

```
data Grafo v = G [v] (v -> [v])
```

Representación de grafos

- `ej_grafo` es la representación del grafo




```
ej_grafo = G [1..5] suc
    where suc 1 = [2,3]
          suc 2 = [4]
          suc 3 = [4]
          suc _ = []
```

Búsqueda de los caminos en un grafo

- `caminosDesde g o te vis` es la lista de los caminos en el grafo g desde el vértice origen o hasta vértices finales (i.e los que verifican el test de encontrado `te`) sin volver a pasar por los vértices visitados `vis`. Por ejemplo,

```
caminosDesde ej_grafo 1 (==4) [] ⇝ [[4,2,1],[4,3,1]]
```

```
caminosDesde :: Eq a => Grafo a -> a -> (a -> Bool) -> [a]
caminosDesde g o te vis
    | te o      = [o:vis]
    | otherwise = concat [caminosDesde g o' te (o:vis)
                          | o' <- suc o,
                            notElem o' vis]
    where G _ suc = g
```

Búsqueda de un camino en un grafo

- `camino g u v` es un camino (i.e una lista de vértices tales que cada uno es un sucesor del anterior) en el grafo `g` desde el vértice `u` al `v`. Por ejemplo,

  ```
  camino ej_grafo 1 4 ⇝ [4,2,1]
  ```

```
camino :: Eq a => Grafo a -> a -> a -> [a]
camino g u v = head (caminosDesde g u (== v) [])
```

Bibliografía

1. H. C. Cunningham (2007) *Notes on Functional Programming with Haskell*.
2. J. Fokker (1996) *Programación funcional*.
3. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J. Gallardo (2004). *Razonando con Haskell (Un curso sobre programación funcional)*.
4. S. Thompson (1999) *Haskell: The Craft of Functional Programming*.
5. E.P. Wentworth (1994) *Introduction to Funcional Programming*.