Programación declarativa (2007–08)

Tema 3: Estructuras de datos

José A. Alonso Jiménez

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

Tema 3: Estructuras de datos (I)

1. Listas
    - Construcción de listas
    - Funciones sobre listas
    - Funciones de orden superior sobre listas
    - Ordenación de listas

2. Listas especiales
    - Cadenas
    - Caracteres
    - Funciones de cadenas y caracteres
    - Listas infinitas y evaluación perezosa

3. Tuplas
    - Uso de tuplas
    - Listas y tuplas
    - Tuplas y currificación

Tema 3: Estructuras de datos (II)

4. Tipos de datos definidos
- Definiciones de tipos
- Números racionales
- Tipo de dato definido: Árboles
- Árboles de búsqueda
- Usos especiales de definiciones de datos

Tema 3: Estructuras de datos

1. Listas
   - Construcción de listas
   - Funciones sobre listas
   - Funciones de orden superior sobre listas
   - Ordenación de listas

2. Listas especiales

3. Tuplas

4. Tipos de datos definidos

Listas

- Las listas se usan para agrupar varios elementos.
- Los elementos de una lista tienen que ser del mismo tipo.
- El tipo de una lista se indica escribiendo el tipo de sus elementos entre corchetes.
- Maneras de construir listas:
  - enumeración
  - con el operador (:)
  - intervalos numéricos
  - por comprensión

Construcción de listas por enumeración

- Ejemplos de listas por enumeración:

```
[1,2,3]                :: [Int]
[True,False,True]      :: [Bool]
[sin, cos, tan]        :: [Float -> Float]
[[1,2,3],[1,4]]        :: [[Int]]
```

- Ejemplos de listas con expresiones:

```
[1+2, 3*4, length [1,2,4,5]]        :: [Int]
[3<4, 2+3==4+1, (2<3) && (2>3)]     :: [Bool]
[map (1+), filter (2<)]             :: [[Int] -> [Int]]
```

- Ejemplos de listas unitarias:

```
[3] :: [Int]
[[True, 3<2]] :: [[Bool]]
```

Construcción de listas por enumeración

La lista vacía es polimorfa:

```
:set +t
tail [3]     ⇝ [] :: [Integer]
tail [True] ⇝ [] :: [Bool]

:t length    ⇝ length :: [a] -> Int
length []    ⇝ 0 :: Int
:t sum       ⇝ sum :: Num a => [a] -> a
sum []       ⇝ 0 :: Integer
:t and       ⇝ and :: [Bool] -> Bool
and []       ⇝ True :: Bool
```

Construcción de listas con el operador :

- `x:l` es la lista obtenida añadiendo el elemento `x` al principio de la lista `l`.
- El tipo de `(:)` es `a -> [a] -> [a]`
- El operador `(:)` asocia por la derecha.
- Ejemplos:

```
1:2:3:[] ⇝ [1,2,3]
1:[2,3]  ⇝ [1,2,3]
```

Construcción de listas mediante intervalos numéricos

- Ejemplos de construcción de listas mediante intervalos numéricos:

```
[3..7]      ⇝ [3,4,5,6,7]
[1,3..10]   ⇝ [1,3,5,7,9]
[7..3]      ⇝ []
[7,6..3]    ⇝ [7,6,5,4,3]
[2.5..6.5]  ⇝ [2.5,3.5,4.5,5.5,6.5]
[2.5..6.3]  ⇝ [2.5,3.5,4.5,5.5,6.5]
[2.5..6.7]  ⇝ [2.5,3.5,4.5,5.5,6.5]
```

- `[n..m]` es la lista de los números desde `n` a `m` de 1 en 1.
- `[n,p..m]` es la lista de los números desde `n` a `m` de `d` en `d` donde `d = p-n`.

Construcción de listas por comprensión

- Ejemplos de lista intensional con un generador:

```
[x*x | x <- [1,3,7] ] ⇝ [1,9,49]
[2*x | x <- [1..10] ] ⇝ [2,4,6,8,10,12,14,16,18,20]
```

- Ejemplos de lista intensional con dos generadores:

```
Main> [[x,y] | x <- [1,2], y <- [3,4]]
[[1,3],[1,4],[2,3],[2,4]]
```

- Ejemplos de lista intensional con patrones:

```
[x+y | [x,y] <- [[1,2],[3,4],[5,6]]] ⇝ [3,7,11]
```

- Ejemplos de lista intensional con restricciones:

```
[x | (x:y) <- [[7,2,3],[1,3,4],[9,6]], x>sum y] ⇝ [7,9]
```

Ejemplos de definiciones con listas por comprensión

- `todosPares xs` se verifica si todos los elementos de la lista `xs` son pares. Por ejemplo,

  ```
  todosPares [2,4,6]    ~> True
  todosPares [2,4,6,7]  ~> False
  ```

  ```haskell
  todosPares :: [Int] -> Bool
  todosPares xs = (xs == [x | x<-xs, even x])
  ```

- `triangulares n` es la lista de las lista de números consecutivos desde `[1]` hasta `[1,2,...,n]`. Por ejemplo,

  ```
  triangulares 4 ~> [[1],[1,2],[1,2,3],[1,2,3,4]]
  ```

  ```haskell
  triangulares :: Int -> [[Int]]
  triangulares n = [[1..x] | x <- [1..n]]
  ```

Tema 3: Estructuras de datos

1. Listas
   - Construcción de listas
   - Funciones sobre listas
   - Funciones de orden superior sobre listas
   - Ordenación de listas

2. Listas especiales

3. Tuplas

4. Tipos de datos definidos

Funciones sobre listas

- Patrón para definiciones sobre listas:
  - Base: `[]`
  - Paso: `x:xs`
- Funciones sobre listas del preludio definidas anteriormente:
  - `head l` es la cabeza de la lista `l`
  - `tail l` es el resto de la lista `l`
  - `sum l` es la suma de los elementos de `l`
  - `length l` es el número de elementos de `l`
  - `map f l` es la lista obtenida aplicando `f` a cada elemento de `l`
  - `filter p l` es la lista de los elementos de `l` que cumplen la propiedad `p`

Comparación y ordenación lexicográfica de listas

- Se pueden comparar y ordenar listas (con operadores como == y <) con la condición de que se puedan o comparar y ordenar sus elementos.
- Definición de la igualdad de listas:

Prelude
```
[]     == []     =  True
(x:xs) == (y:ys) =  x==y && xs==ys
_      == _      =  False
```

Concatenación de listas

- `l1 ++ l2` es la concatenación de `l1` y `l2`. Por ejemplo,

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

Prelude

```haskell
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

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

```
concat [[1,2,3],[4,5],[],[1,2]] ⇝ [1,2,3,4,5,1,2]
```

Prelude

```haskell
concat :: [[a]] -> [a]
concat []     = []
concat (x:xs) = x ++ concat xs
```

Propiedades de la concatenación de listas

- Longitud de la concatenación:

```
prop_length_append :: [Int] -> [Int] -> Bool
prop_length_append xs ys =
    length(xs++ys) == (length xs) + (length ys)
```

```
Main> quickCheck prop_length_append
OK, passed 100 tests.
```

- Longitud de la concatenación general:

```
prop_length_concat :: [[Int]] -> Bool
prop_length_concat xss =
    length(concat xss) == sum (map length xss)
```

```
Main> quickCheck prop_length_concat
OK, passed 100 tests.
```

Selección de partes de una lista

- `head l` es la cabeza de la lista `l`. Por ejemplo,

```
head [3,5,2] ⇝ 3
```

Prelude

```haskell
head :: [a] -> a
head (x:_) = x
```

- `tail l` es el resto de la lista `l`. Por ejemplo,

```
tail [3,5,2]     ⇝ [5,2]
```

Prelude

```haskell
tail :: [a] -> [a]
tail (_:xs) = xs
```

Selección de partes de una lista

- Conjetura:

```haskell
prop_head_tail_1 :: [Int] -> Bool
prop_head_tail_1 xs =
    (head xs) : (tail xs) == xs
```

- Comprobación de la conjetura:

```
quickCheck prop_head_tail_1
Falsifiable, after 2 tests:
[]
```

- Comprobación del contraejemplo:

```
Main> (head []) : (tail []) == []
False
Main> head []
Program error: pattern match failure: head []
```

Selección de partes de una lista

- Propiedad:

```
prop_head_tail :: [Int] -> Property
prop_head_tail xs =
    not (null xs) ==> (head xs) : (tail xs) == xs
```

- Comprobación de la propiedad:

```
Main> quickCheck prop_head_tail
OK, passed 100 tests.
```

Selección de partes de una lista

- `last l` es el último elemento de la lista `l`. Por ejemplo,

```
last [1,2,3] ⇝ 3
```

Prelude

```haskell
last :: [a] -> a
last [x]    = x
last (_:xs) = last xs
```

- `init l` es la lista `l` sin el último elemento. Por ejemplo,

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

Prelude

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

Selección de partes de una lista

- Propiedad:

```
prop_init_last :: [Int] -> Property
prop_init_last xs =
    not (null xs) ==>
    (init xs) ++ [last xs] == xs
```

- Comprobación:

```
Main> quickCheck prop_init_last
OK, passed 100 tests.
```

Selección de partes de una lista

- `take n l` es la lista de los `n` primeros elementos de `l`. Por ejemplo,

```
take 2 [3,5,4,7]  ⇝ [3,5]
take 12 [3,5,4,7] ⇝ [3,5,4,7]
```

Prelude

```
take :: Int -> [a] -> [a]
take n _   | n <= 0  = []
take _ []            = []
take n (x:xs)        = x : take (n-1) xs
```

Selección de partes de una lista

- Propiedad:

```
prop_take :: Int -> Int -> [Int] -> Property
prop_take n m xs =
    n >= 0 && m >= 0 ==>
    take n (take m xs) == take (min n m) xs
```

- Comprobación:

```
Main> quickCheck prop_take
OK, passed 100 tests.
```

Selección de partes de una lista

- `drop n l` es la lista obtenida eliminando los primeros `n` elementos de la lista `l`. Por ejemplo,

```
drop 2 [3..10]  ⇝ [5,6,7,8,9,10]
drop 12 [3..10] ⇝ []
```

Prelude

```haskell
drop :: Int -> [a] -> [a]
drop n xs | n <= 0  = xs
drop _ []           = []
drop n (_:xs)       = drop (n-1) xs
```

Selección de partes de una lista

- Propiedad de drop:

```
prop_drop :: Int -> Int -> [Int] -> Property
prop_drop n m xs =
    n >= 0 && m >= 0 ==>
    drop n (drop m xs) == drop (n+m) xs
```

- Propiedad de take y drop:

```
prop_take_drop :: Int -> [Int] -> Bool
prop_take_drop n xs =
    (take n xs) ++ (drop n xs) == xs
```

Selección de partes de una lista

- `l !! n` es elemento `n`–ésimo de `l`, empezando a numerar con el 0. Por ejemplo,

  `[1,3,2,4,9,7] !! 3` ⇝ `4`

Prelude

```
infixl 9  !!

(!!) :: [a] -> Int -> a
(x:_)  !! 0 = x
(_:xs) !! n = xs !! (n-1)
```

Inversa de una lista

- `reverse l` es la inversa de `l`. Por ejemplo,

```
reverse [1,4,2,5] ⇝ [5,2,4,1]
```

Prelude

```haskell
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

- Propiedades:

```haskell
prop_reverse :: [Int] -> [Int] -> Bool
prop_reverse xs ys =
    reverse(xs ++ ys) == (reverse ys) ++ (reverse xs)
    && reverse(reverse xs) == xs
```

Longitud de una lista

- `length l` es el número de elementos de `l`. Por ejemplo,

  `length [1,3,6]` $\rightsquigarrow$ `3`

Prelude
```
length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs
```

Test de pertenencia a una lista

- `elem e l` se verifica si `e` es un elemento de `l`. Por ejemplo,

```
elem 2 [1,2,3] ⇝ True
elem 4 [1,2,3] ⇝ False
```

- Definición recursiva

```
elem_1 :: Eq a => a -> [a] -> Bool
elem_1 _ []       = False
elem_1 x (y:ys) = (x==y) || elem_1 x ys
```

- Definición con `or` y lista por comprensión:

```
elem_2 :: Eq a => a -> [a] -> Bool
elem_2 x ys = or [x==y | y <- ys]
```

Test de pertenencia a una lista

- `notElem e l` se verifica si `e` es un elemento de `l`. Por ejemplo,

```
notElem 2 [1,2,3] ⇝ False
notElem 4 [1,2,3] ⇝ True
```

- Definición recursiva

```
notElem_1 :: Eq a => a -> [a] -> Bool
notElem_1 _ []     = True
notElem_1 x (y:ys) = (x/=y) && notElem_1 x ys
```

- Definición con `or` y lista por comprensión:

```
notElem_2 :: Eq a => a -> [a] -> Bool
notElem_2 x ys = and [x/=y | y <- ys]
```

Test de pertenencia a una lista

- Equivalencia de las definiciones:

```
prop_equivalencia_notElem :: Int -> [Int] -> Bool
prop_equivalencia_notElem x ys =
    notElem_1 x ys == notElem x ys &&
    notElem_2 x ys == notElem x ys

prop_equivalencia_elem :: Int -> [Int] -> Bool
prop_equivalencia_elem x ys =
    elem_1 x ys == elem x ys &&
    elem_2 x ys == elem x ys
```

Test de pertenencia a una lista

- Propiedad de `elem` y `notElem`:

```
prop_elem_notElem :: Int -> [Int] -> Bool
prop_elem_notElem x ys =
    (elem x ys || notElem x ys) &&
    not ((elem x ys) && (notElem x ys))
```

Tema 3: Estructuras de datos

1. Listas
   - Construcción de listas
   - Funciones sobre listas
   - Funciones de orden superior sobre listas
   - Ordenación de listas

2. Listas especiales

3. Tuplas

4. Tipos de datos definidos

Funciones de orden superior sobre listas: map

- `map f l` es la lista obtenida aplicando `f` a cada elemento de `l`. Por ejemplo,

  ```
  map (2*) [1,2,3] ⇝ [2,4,6]
  ```

- Definición por comprensión:

  Prelude
  ```
  map :: (a -> b) -> [a] -> [b]
  map f xs = [ f x | x <- xs ]
  ```

- Definición recursiva:

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

Funciones de orden superior sobre listas: map

- `filter p l` es la lista de los elementos de `l` que cumplen la propiedad `p`. Por ejemplo,

  ```
  filter even [1,3,5,4,2,6,1] ⇝ [4,2,6]
  filter (>3) [1,3,5,4,2,6,1] ⇝ [5,4,6]
  ```

- Definición por comprensión:

Prelude

```haskell
filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [ x | x <- xs, p x ]
```

- Definición por recursión:

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

Ejemplo de abstración de patrones

- Ejemplos de definiciones con el mismo patrón:
  - `sum l` es la suma de los elementos de `l`. Por ejemplo,

    ```
    sum [3,4,5,6] ⇝ 18
    ```

    Prelude

    ```haskell
    sum []     = 0
    sum (x:xs) = x + sum xs
    ```

  - `product l` es el producto de los elementos de `l`. Por ejemplo,

    ```
    product [2,3,5] ⇝ 30
    ```

    Prelude

    ```haskell
    product []     = 1
    product (x:xs) = x * product xs
    ```

Ejemplo de abstración de patrones

- Ejemplos de definiciones con el mismo patrón:
  - `and l` se verifica si todos los elementos de `l` son verdaderos. Por ejemplo,

    ```
    and [1<2, 2<3, 1 /= 0]  ⇝  True
    and [1<2, 2<3, 1 == 0]  ⇝  False
    ```

    Prelude

    ```haskell
    and []     = True
    and (x:xs) = x && and xs
    ```

  - `or l` se verifica si algún elemento de `l` es verdadero. Por ejemplo,

    ```
    or [2>3, 5<9]  ⇝  True
    or [2>3, 9<5]  ⇝  False
    ```

    Prelude

    ```haskell
    or []     = False
    or (x:xs) = x || or xs
    ```

Ejemplo de abstración de patrones

- Patrón:
  `foldr op e l` pliega por la derecha la lista `l` colocando el operador op entre sus elementos y el elemento e al final. Es decir,

```
foldr f e [x1,x2,x3]      ⇝ x1 f (x2 f (x3 f e))
foldr f e [x1,x2,...,xn]  ⇝ x1 f (x2 f (... f (xn f e)))
```

  Por ejemplo,

```
foldr (+) 3 [2,3,5] ⇝ 13
foldr (-) 3 [2,3,5] ⇝ 1
```

Prelude

```haskell
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f e []      = e
foldr f e (x:xs)  = f x (foldr f e xs)
```

Ejemplo de abstración de patrones

- Redefiniciones con el patrón:

```
sum_1     = foldr (+) 0
product_1 = foldr (*) 1
and_1     = foldr (&&) True
or_1      = foldr (||) False
```

- Definición del factorial mediante plegados:
  fact n es el factorial de n. Por ejemplo,

  | fact 5 ⇝ 120

```
fact n = foldr (*) 1 [1..n]
```

Ejemplo de abstración de patrones

- Plegado por la izquierda:
  `foldl op e l` pliega por la izquierda la lista `l` colocando el operador `op` entre sus elementos y el elemento `e` al principio. Es decir,

  ```
  foldl f e [x1,x2,x3]      ⇝ (((e f x1) f x2) f x3
  foldl f e [x1,x2,...,xn] ⇝ (...((e f x1) f x2) ... f xn
  ```

  Por ejemplo,

  ```
  foldl (+) 3 [2,3,5] ⇝ 13
  foldl (-) 3 [2,3,5] ⇝ -7
  ```

  Prelude

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

Plegado acumulativo: `scanr`

- `scanr op e l` pliega por la derecha la lista `l` colocando el operador `op` entre sus elementos y el elemento `e` al final y escribe los resultados acumulados. Es decir,

```
scanr op e [x1,x2,x3] ⇝ [x1 op (x2 op (x3 op e)),
                                x2 op (x3 op e),
                                       x3 op e,
                                             e]
```

  Por ejemplo,

```
scanr (+) 3 [2,3,5] ⇝[13,11,8,3]
```

Prelude

```
scanr :: (a -> b -> b) -> b -> [a] -> [b]
scanr f q0 []     = [q0]
scanr f q0 (x:xs) = f x q : (q:qs)
                    where (q:qs) = scanr f q0 xs
```

Plegado acumulativo: scanl

- `scanl` es análogo a `scanr` pero empezando por la izquierda. Por ejemplo,

  `scanl (+) 3 [2,3,5]` ⇝ `[3,5,8,13]`

- `factoriales n` es la lista de los factoriales desde el factorial de 0 hasta el factorial de `n`. Por ejemplo,

  `factoriales 5` ⇝ `[1,1,2,6,24,120]`

```
factoriales :: Int -> [Int]
factoriales n = scanl (*) 1 [1..n]
```

Segmentos iniciales: takeWhile

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

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

Prelude

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

Segmentos finales: `dropWhile`

- `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

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

Tema 3: Estructuras de datos

1. Listas
   - Construcción de listas
   - Funciones sobre listas
   - Funciones de orden superior sobre listas
   - Ordenación de listas

2. Listas especiales

3. Tuplas

4. Tipos de datos definidos

Ordenación por inserción

- `inserta e l` inserta el elemento `e` en la lista `l` delante del primer elemento de `l` mayor o igual que `e`. Por ejemplo,

  `inserta 5 [2,4,7,3,6,8,10]` ⇝ `[2,4,5,7,3,6,8,10]`

```
inserta          :: Ord a => a -> [a] -> [a]
inserta e []      = [e]
inserta e (x:xs)
    | e<=x        = e:x:xs
    | otherwise   = x : inserta e xs
```

Ordenación por inserción

- `ordena_por_inserción l` es la lista `l` ordenada mediante inserción, Por ejemplo,

  ```
  ordena_por_inserción [2,4,3,6,3] ⇝ [2,3,3,4,6]
  ```

- Definición recursiva:

```haskell
ordena_por_inserción :: Ord a => [a] -> [a]
ordena_por_inserción []     = []
ordena_por_inserción (x:xs) =
    inserta x (ordena_por_inserción xs)
```

- Definición por plegado por la derecha:

```haskell
ordena_por_inserción_1 :: Ord a => [a] -> [a]
ordena_por_inserción_1  = foldr inserta []
```

Ordenación por inserción

- Definición por plegado por la izquierda:

```
ordena_por_inserción_2 :: Ord a => [a] -> [a]
ordena_por_inserción_2  = foldl (flip inserta) []
```

- La última es la más eficiente:

```
ordena_por_inserción   [100,99..1] (51959 reductions, 68132 cells)
ordena_por_inserción_1 [100,99..1] (51960 reductions, 68034 cells)
ordena_por_inserción_2 [100,99..1] ( 3451 reductions,  5172 cells)
```

Ordenación por inserción

- `mínimo l` es el menor elemento de la lista `l`. Por ejemplo,

  `mínimo [3,2,5]` ⇝ `2`

```
mínimo l = head (ordena_por_inserción l)
```

La complejidad de `mínimo` es lineal:

```
mínimo [10,9..1]                  (      300 red)
mínimo [100,99..1]                (     2550 red)
mínimo [1000,999..1]              (    25050 red)
mínimo [10000,9999..1]            (   250050 red)
```

aunque la complejidad de `ordena_por_inserción` es cuadrática

```
ordena_por_inserción [10,9..1]       (      750 red)
ordena_por_inserción [100,99..1]     (    51960 red)
ordena_por_inserción [1000,999..1]   (  5019060 red)
ordena_por_inserción [10000,9999..1] (500190060 red)
```

Ordenación por inserción

- `lista_ordenada l` se verifica si la lista `l` está ordenada de menor a mayor. Por ejemplo,

```
lista_ordenada [1,3,3,5] ⇝ True
lista_ordenada [1,3,5,3] ⇝ False
```

```haskell
lista_ordenada :: Ord a => [a] -> Bool
lista_ordenada []       = True
lista_ordenada [_]      = True
lista_ordenada (x:y:xs) = x<=y && lista_ordenada (y:xs)
```

- El valor de `ordena_por_inserción` es una lista ordenada

```haskell
prop_ordena_por_inserción_ordenada :: [Int] -> Bool
prop_ordena_por_inserción_ordenada xs =
    lista_ordenada (ordena_por_inserción xs)
```

Ordenación por mezcla

- `mezcla l1 l2` es la lista ordenada obtenida al mezclar las listas ordenadas `l1` y `l2`. Por ejemplo,

  `mezcla [1,3,5] [2,9]` $\rightsquigarrow$ `[1,2,3,5,9]`

```
mezcla :: Ord a => [a] -> [a] -> [a]
mezcla [] ys           = ys
mezcla xs []           = xs
mezcla (x:xs) (y:ys)
    | x <= y           = x : mezcla xs (y:ys)
    | otherwise        = y : mezcla (x:xs) ys
```

Ordenación por mezcla

- `ordena_por_m l` es la lista `l` ordenada mediante mezclas, Por ejemplo,

```
ordena_por_m [2,4,3,6,3] ⇝ [2,3,3,4,6]
```

```
ordena_por_m :: Ord a => [a] -> [a]
ordena_por_m []  = []
ordena_por_m [x] = [x]
ordena_por_m xs  =
    mezcla (ordena_por_m ys) (ordena_por_m zs)
    where medio = (length xs) `div` 2
          ys    = take medio xs
          zs    = drop medio xs
```

Ordenación por mezcla

- El valor de `ordena_por_m` es una lista ordenada

```
prop_ordena_por_mezcla_ordenada :: [Int] -> Bool
prop_ordena_por_mezcla_ordenada xs =
    lista_ordenada (ordena_por_m xs)
```

Ordenación rápida ("quicksort")

`ordena R xs` es la lista `xs` ordenada mediante el procedimiento de ordenación rápida. Por ejemplo,

```
ordenaR [5,2,7,7,5,19,3,8,6] ⇝ [2,3,5,5,6,7,7,8,19]
```

```haskell
ordenaR :: Ord a => [a] -> [a]
ordenaR []     = []
ordenaR (x:xs) = ordenaR menores ++ [x] ++ ordenaR mayores
                 where menores = [e | e<-xs, e<x]
                       mayores = [e | e<-xs, e>=x]
```

Tema 3: Estructuras de datos

1. Listas

2. Listas especiales
   - Cadenas
   - Caracteres
   - Funciones de cadenas y caracteres
   - Listas infinitas y evaluación perezosa

3. Tuplas

4. Tipos de datos definidos

Cadenas

- Las cadenas son listas cuyos elementos son caracteres.
- Las cadenas se anotan entre comillas.
- Ejemplos de cadenas: `"lunes"`, `"Juan Luis"`
- A las cadenas se le pueden aplicar las funciones sobre listas. Por ejemplo,

```
length "martes"   ⇝ 6
"mar" ++ "tes"    ⇝ "martes"
"hola" < "mundo"  ⇝ True
```

- El tipo de las cadenas es `String`.

Tema 3: Estructuras de datos

1. Listas

2. Listas especiales
   - Cadenas
   - Caracteres
   - Funciones de cadenas y caracteres
   - Listas infinitas y evaluación perezosa

3. Tuplas

4. Tipos de datos definidos

Caracteres

- Los caracteres pueden ser letras, cifras y signos de puntuación.
- Los caracteres se anotas entre apóstrofos.
- Ejemplos de caracteres: 'a', '+'
- El tipo de los caracteres es `Char`.
- Una lista de caracteres es una cadena. Por ejemplo,

```
['h','o','l','a'] ⇝ "hola"
head "hola"       ⇝ 'h'
```

Tema 3: Estructuras de datos

1. Listas

2. Listas especiales
   - Cadenas
   - Caracteres
   - Funciones de cadenas y caracteres
   - Listas infinitas y evaluación perezosa

3. Tuplas

4. Tipos de datos definidos

Funciones de transformación entre cadenas y enteros

- `ord c` es el código ASCII del carácter `c`.
- `chr n` es el carácter de código ASCII el entero `n`.
- Ejemplos:

```
:load Hugs.Char
ord 'A' ⇝ 65
chr 65  ⇝ 'A'
```

- Para usar las funciones sobre caracteres en programas hay que importar el módulo `Char` escribiendo al principio del fichero

```
import Hugs.Char
```

Funciones reconocedoras predefinidas

- Funciones:

`isSpace x`	x es un espacio
`isUpper x`	x está en mayúscula
`isLower x`	x está en minúscula
`isAlpha x`	x es un carácter alfabético
`isDigit x`	x es un dígito
`isAlphaNum x`	x es un carácter alfanumérico

- Ejemplos de definición:

Prelude

```
isDigit c   =  c >= '0' && c <= '9'
```

Funciones sobre caracteres

- `dígitoDeCarácter c` es el dígito correspondiente al carácter numérico `c`. Por ejemplo,

  ```
  dígitoDeCarácter '3' ⇝ 3
  ```

```haskell
dígitoDeCarácter :: Char -> Int
dígitoDeCarácter c = ord c - ord '0'
```

Definición alternativa

```haskell
dígitoDeCarácter' :: Char -> Int
dígitoDeCarácter' c
    | isDigit c = ord c - ord '0'
```

Funciones sobre caracteres

- `carácterDeDígito n` es el carácter correspondiente al dígito `n`. Por ejemplo,

  ```
  carácterDeDígito 3 ⇝ '3'
  ```

```haskell
carácterDeDígito   :: Int -> Char
carácterDeDígito n  = chr (n + ord '0')
```

Convesiones entre minúsculas y mayúsculas:

- `toUpper c` es el carácter c en mayúscula.
- `toLower c` es el carácter c en minúcula.
- Ejemplos:

```
toUpper 'a'                ⇝ 'A'
toLower 'A'                ⇝ 'a'
map toUpper "sevilla"      ⇝ "SEVILLA"
map toLower "SEVILLA"      ⇝ "sevilla"
```

Funciones del preludio específicas para cadenas

- `words c` es la lista de palabras de la cadena `c`. Por ejemplo,

```
Main> words "esto es una cadena"
["esto","es","una","cadena"]
```

- `lines c` es la lista de líneas de la cadena `c`. Por ejemplo,

```
Main> lines "primera linea\ny segunda"
["primera linea","y segunda"]
Main> words "primera linea\ny segunda"
["primera","linea","y","segunda"]
```

- `unwords` es la inversa de `words`. Por ejemplo,

```
Main> unwords ["esto","es","una","cadena"]
"esto es una cadena"
```

- `unlines` es la inversa de `lines`. Por ejemplo,

```
Main> unlines ["primera linea","y segunda"]
"primera linea\ny segunda\n"
```

Tema 3: Estructuras de datos

1. Listas

2. Listas especiales
   - Cadenas
   - Caracteres
   - Funciones de cadenas y caracteres
   - Listas infinitas y evaluación perezosa

3. Tuplas

4. Tipos de datos definidos

Listas infinitas

- `desde n` es la lista de los números enteros a partir de `n`. Por ejemplo,

  ```
  desde 5 ⇝ [5,6,7,8,9,10,11,12,13,14,{Interrupted!}
  ```

  se interrumpe con `Control-C`.

  ```
  desde :: Int -> [Int]
  desde n = n : desde (n+1)
  ```

  Definición alternativa

  ```
  desde' :: Int -> [Int]
  desde' n = [n..]
  ```

Cálculos con listas infinitas

- Calcular la lista de los 10 primeros sucesores de 7:

```
Main> tail (take 11 (desde 7))
[8,9,10,11,12,13,14,15,16,17]
Main> tail (take 11 [7..])
[8,9,10,11,12,13,14,15,16,17]
```

- Calcular las potencias de 2 menores que 1000:

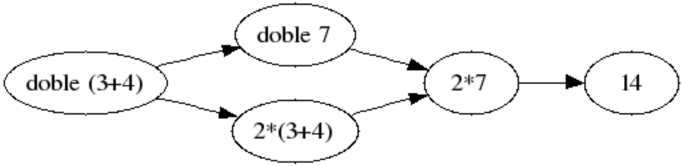
```
Main> takeWhile (<1000) (map (2^) (desde 1))
[2,4,8,16,32,64,128,256,512]
Main> takeWhile (<1000) (map (2^) [1..])
[2,4,8,16,32,64,128,256,512]
```

Evaluación perezosa

Se considera la siguiente definición

```
doble :: Int -> Int
doble x = 2*x
```

Entonces, las posibles evaluaciones de `doble (3+4)` son

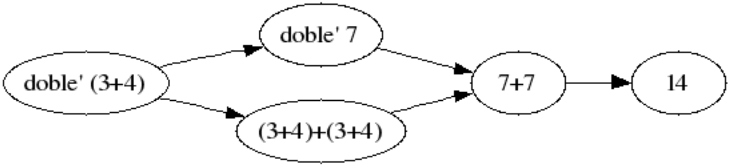


Evaluación perezosa

Se considera la siguiente definición

```
doble' :: Int -> Int
doble' x = x+x
```

Entonces, las posibles evaluaciones de `doble' (3+4)` son



Notar que en la rama superior, 3+4 sólo se evalúa una vez.

Métodos de evaluación

- evaluación perezosa (en inglés "lazy evaluation"): se calcula una expresión parcial si realmente se necesita el valor para calcular el resultado.
- evaluación voraz (en inglés "eager evaluation"): se calcula el valor de los parámetros y se aplica la función a sus valores.
- Haskell usa evaluación perezosa.
- LISP, Scheme y ML usan evaluación voraz.

Ejemplo de ventaja de la evaluación perezosa

En el tema anterior, se definió `primo` mediante

```
primo x = divisores x == [1,x]
```

Al aplicar la definición sólo se calculan los dos primeros primos. Por ejemplo,

```
ED> :set +s
ED> primo 30
False
(108 reductions, 177 cells)
ED> primo 30000
False
(108 reductions, 177 cells)
```

Explicación del comportamiento perezoso de == y &&

- La regla recursiva de la definición de == es
  `(x:xs) == (y:ys) =  x==y && xs==ys`
  Si `x==y` es falsa, entonces también lo es `(x:xs) == (y:ys)`.
- La definición de && es

Prelude
```
False && x    = False
True  && x    = x
```

Si el primer parámetro es falso, entonces la conjunción es falsa.

Cálculos infinitos

Las funciones que necesitan todos los elementos de una lista, no pueden aplicarse a listas infinitas. Por ejemplo,

```
Main> length (desde 1)
ERROR - Garbage collection fails to reclaim sufficient space

Main> head (desde 1)
1

Main> (desde 1) ++ [3]
[1,2,3,4,5,6,7,8,9,10,11,12,,...Interrupted!

Main> last (desde 1)
ERROR - Garbage collection fails to reclaim sufficient space
```

Funciones sobre listas infinitas definidas en el preludio

- `[n..]` es equivalente a `desde n`.
- `repeat x` es una lista infinita con el único elemento `x`. Por ejemplo,

```
Main> repeat 'a'
"aaaaaaaaaaaaaaaaaaaaaaaaaa{Interrupted!}
```

```
repeat' :: a -> [a]
repeat' x = x : repeat' x
```

Una definición alternativa es

```
Prelude
repeat :: a -> [a]
repeat x  = xs where xs = x:xs
```

Funciones sobre listas infinitas definidas en el preludio

- `replicate n x` es una lista con `n` copias del elemento `x`. Por ejemplo,

```
Main> replicate 10 3
[3,3,3,3,3,3,3,3,3,3]
```

Prelude

```haskell
replicate :: Int -> a -> [a]
replicate n x = take n (repeat x)
```

Funciones sobre listas infinitas definidas en el preludio

- `iterate f x` es la lista cuyo primer elemento es `x` y los siguientes elementos se calculan aplicando la función `f` al elemento anterior. Por ejemplo,

```
Main> iterate (+1) 3
[3,4,5,6,7,8,9,10,11,12,{Interrupted!}
Main> iterate (*2) 1
[1,2,4,8,16,32,64,{Interrupted!}
Main> iterate (`div` 10) 1972
[1972,197,19,1,0,0,0,0,0,0,{Interrupted!}
```

Prelude

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

Presentación de un número como cadena

- `deEnteroACadena n` es la cadena correspondiente al número entero `n`. Por ejemplo, `deEnteroACadena 1972` ⇝ `"1972"`

```
deEnteroACadena :: Int -> String
deEnteroACadena = map carácterDeDígito
                  . reverse
                  . map (`rem` 10)
                  . takeWhile (/= 0)
                  . iterate (`div` 10)
```

Ejemplo de cálculo

```
iterate (`div` 10) 1972                          ⇝ [1972,197,19,1,0,0,0,
(takeWhile (/= 0) . iterate (`div` 10)) 1972     ⇝ [1972,197,19,1]
map (`rem` 10) [1972,197,19,1]                   ⇝ [2,7,9,1]
reverse [2,7,9,1]                                ⇝ [1,9,7,2]
map carácterDeDígito [1,9,7,2]                   ⇝ "1972"
```

Presentación de un número como cadena

- La función `deEnteroACadena` es un caso particular de `show`. Por ejemplo,

```
show 1972 ⇝ "1972"
```

Cálculo perezoso de la lista de los números primos

- `primos_por_criba` es la lista de los números primos mediante la criba de Erastótenes.

```
Main> primos_por_criba
[2,3,5,7,11,13,17,19,23,29,{Interrupted!}
Main> take 10 primos_por_criba
[2,3,5,7,11,13,17,19,23,29]
```

```
primos_por_criba :: [Int]
primos_por_criba = map head (iterate eliminar [2..])
    where eliminar (x:xs) = filter (no_multiplo x) xs
          no_multiplo x y = y `mod` x /= 0
```

Cálculo perezoso de la lista de los números primos

- Para ver el cálculo, consideramos la siguiente variación

```
primos_por_criba_aux =
    map (take 10) (iterate eliminar [2..])
    where eliminar (x:xs) = filter (no_multiplo x) xs
          no_multiplo x y = y `mod` x /= 0
```

Entonces,

```
Main> take 5 primos_por_criba_aux
[[ 2, 3, 4, 5, 6, 7, 8, 9,10,11],
 [ 3, 5, 7, 9,11,13,15,17,19,21],
 [ 5, 7,11,13,17,19,23,25,29,31],
 [ 7,11,13,17,19,23,29,31,37,41],
 [11,13,17,19,23,29,31,37,41,43]]
```

Cálculo perezoso de la lista de los números primos

- Definición con lista por comprensión:

```
primos_por_criba' :: [Int]
primos_por_criba' = criba [2..]
    where criba (p:xs) = p : criba [n | n<-xs,
                                        n `mod` p /= 0]
```

Tema 3: Estructuras de datos

1. Listas

2. Listas especiales

3. Tuplas
   - Uso de tuplas
   - Listas y tuplas
   - Tuplas y currificación

4. Tipos de datos definidos

Tuplas

- Una tupla consiste en un número fijo de valores, que están agrupados como una entidad.
- Ejemplos de uso de tuplas en modelizaciones:

punto	`(3.5,4.2)`	`::`	`(Float, Float)`
teléfono	`("Ana Pi", 954213465)`	`::`	`(String, Int)`
precio	`("periódico", 1.9)`	`::`	`(String, Float)`
vacía	`()`	`::`	`()`

- Los valores pueden ser de diferentes tipos.
- Las tuplas se anotan con paréntesis.
- Clases de tuplas:
  - Un par es una tupla con dos elementos.
  - Una terna es una tupla con tres elementos.
  - La tupla vacía es la tupla sin elementos.

Funciones del preludio sobre tuplas

- `fst p` es la primera componente del par `p`. Por ejemplo,

  ```
  fst (3,2) ⇝ 3
  ```

  Prelude

  ```haskell
  fst :: (a,b) -> a
  fst (x,_) = x
  ```

- `snd p` es la segunda componente del par `p`. Por ejemplo,

  ```
  snd (3,2) ⇝ 2
  ```

  Prelude

  ```haskell
  snd :: (a,b) -> b
  snd (_,y) = y
  ```

Definición de funciones sobre tuplas

- `fst3 t` es la primera componente de la terna `t`.
- `snd3 t` es la segund componente de la terna `t`.
- `thd3 t` es la tercera componente de la terna `t`.
- Ejemplos:

```
fst3 (3,2,5) ⇝ 3
snd3 (3,2,5) ⇝ 2
thd3 (3,2,5) ⇝ 5
```

```
fst3 :: (a,b,c) -> a
fst3 (x,_,_) = x
snd3 :: (a,b,c) -> b
snd3 (_,y,_) = y
thd3 :: (a,b,c) -> c
thd3 (_,_,z) = z
```

Definición de funciones sobre tuplas

- `variable p` es la cadena correspondiente al par `p` formado por un carácter y un número. Por ejemplo,

  ```
  variable ('x',3) ⇝ "x3"
  ```

  ```
  variable (c,n) = [c] ++ show n
  ```

  ```
  Main> :type variable
  variable :: Show a => (Char,a) -> [Char]
  ```

Definición de funciones sobre tuplas

- `distanciaL p` es la distancia del punto `p`, representado mediante listas, al origen. Por ejemplo,

```
distanciaL [3.0,4.0] ⇝ 5.0
```

```haskell
distanciaL :: [Float] -> Float
distanciaL [x,y] = sqrt (x*x+y*y)
```

- `distanciaT p` es la distancia del punto `p`, representado mediante tuplas, al origen. Por ejemplo,

```
distanciaT (3.0,4.0) ⇝ 5.0
```

```haskell
distanciaT :: (Float, Float) -> Float
distanciaT (x,y) = sqrt (x*x+y*y)
```

Ejemplo de uso de tupla para devolver varios resultados

- `splitAt n l` es el par formado por la lista de los `n` primeros elementos de la lista `l` y la lista `l` sin los `n` primeros elementos. Por ejemplo,

```
splitAt 3 [5,6,7,8,9,2,3] ⇝ ([5,6,7],[8,9,2,3])
splitAt 4 "sacacorcho"    ⇝ ("saca","corcho")
```

Prelude

```
splitAt :: Int -> [a] -> ([a], [a])
splitAt n xs | n <= 0  = ([],xs)
splitAt _ []           = ([],[])
splitAt n (x:xs)       = (x:xs',xs'')
    where (xs',xs'') = splitAt (n-1) xs
```

Ejemplo de uso de tupla para devolver varios resultados

- Definición alternativa

```
splitAt_alt :: Int -> [a] -> ([a], [a])
splitAt_alt n xs  = (take n xs, drop n xs)
```

- La definición alternativa es un poco menos eficiente, Por ejemplo,

```
Main> :s +s
Main> splitAt_alt 3 [5,6,7,8,9,2,3]
([5,6,7],[8,9,2,3])
(228 reductions, 330 cells)
Main> splitAt 3 [5,6,7,8,9,2,3]
([5,6,7],[8,9,2,3])
(176 reductions, 283 cells)
```

Ejemplo de uso de tupla para aumentar la eficiencia

- `incmin l` es la lista obtenida añadiendo a cada elemento de `l` el menor elemento de `l`. Por ejemplo,

  ```
  incmin [3,1,4,1,5,9,2,6] ⇝ [4,2,5,2,6,10,3,7]
  ```

```haskell
incmin :: [Int] -> [Int]
incmin l = map (+e) l
           where e                = mínimo l
                 mínimo [x]       = x
                 mínimo (x:y:xs) = min x (mínimo (y:xs))
```

- Con la definición anterior se recorre la lista dos veces: una para calcular el mínimo y otra para sumarlo.

Ejemplo de uso de tupla para aumentar la eficiencia

- Con la siguiente definición la lista se recorre sólo una vez.

```
incmin' :: [Int] -> [Int]
incmin' [] = []
incmin' l = nuevalista
    where (minv, nuevalista) = un_paso l
          un_paso [x]        = (x, [x+minv])
          un_paso (x:xs)     = (min x y, (x+minv):ys)
                               where (y,ys) = un_paso xs
```

Tema 3: Estructuras de datos

1. Listas

2. Listas especiales

3. Tuplas
    - Uso de tuplas
    - Listas y tuplas
    - Tuplas y currificación

4. Tipos de datos definidos

Listas de asociación

- Una lista de asociación es una lista de pares. Los primeros elementos son las claves y los segundos son los valores.
- `buscar z l` es el valor del primer elemento de la lista de asociación `l` cuya clave es `z`. Por ejemplo,

  ```
  buscar 'b' [('a',1),('b',2),('c',3),('b',4)] ⇝ 2
  ```

```haskell
buscar :: Eq a => a -> [(a,b)] -> b
buscar z ((x,y):r)
  | x == z     = y
  | otherwise = buscar z r
```

Funciones de agrupamiento

- `zip x y` es el producto cartesiano de `x` e `y`. Por ejemplo,
  ```
  zip [1,2,3] "abc" ⇝ [(1,'a'),(2,'b'),(3,'c')]
  zip [1,2] "abc"   ⇝ [(1,'a'),(2,'b')]
  ```
- `zipWith f x y` es la lista obtenida aplicando la función `f` a los elementos correspondientes de las listas `x` e `y`. Por ejemplo,
  ```
  zipWith (+) [1,2,3] [4,5,6] ⇝ [5,7,9]
  zipWith (*) [1,2,3] [4,5,6] ⇝ [4,10,18]
  ```

Prelude

```haskell
zipWith :: (a->b->c) -> [a]->[b]->[c]
zipWith f (a:as) (b:bs) = f a b : zipWith f as bs
zipWith _ _      _      = []

zip :: [a] -> [b] -> [(a,b)]
zip = zipWith  (\a b -> (a,b))
```

Funciones de agrupamiento

- Definición alternativa de `zip` sin función anónima:

```
zip_alt_1 = zipWith emparejar
    where emparejar a b = (a,b)
```

- Definición recursiva de `zip`:

```
zip_alt_2 (a:as) (b:bs) = (a,b) : zip_alt_2 as bs
zip_alt_2 _      _      = []
```

Definiciones con `zip`: Posiciones

- `posiciones xs` es la lista de los pares formados por los elementos de `xs` junto con su posición. Por ejemplo,

  ```
  posiciones [3,5,2,5] ⇝ [(3,1),(5,2),(2,3),(5,4)]
  ```

```haskell
posiciones :: [a] -> [(a,Int)]
posiciones xs = zip xs [1..length xs]
```

- `posición x ys` es la posición del elemento `x` en la lista `ys`. Por ejemplo,

  ```
  posición 5 [1,5,3] ⇝ 2
  ```

```haskell
posición :: Eq a => a -> [a] -> Int
posición x xs = buscar x (posiciones xs)
```

Definiciones con `zip`: Fibonacci

- `fibs` es la sucesión de los números de Fibonacci. Por ejemplo,

  `take 10 fibs` ⇝ `[1,1,2,3,5,8,13,21,34,55]`

```haskell
fibs = 1 : 1 : [a+b | (a,b) <- zip fibs (tail fibs)]
```

Tema 3: Estructuras de datos

1. Listas

2. Listas especiales

3. Tuplas
    - Uso de tuplas
    - Listas y tuplas
    - Tuplas y currificación

4. Tipos de datos definidos

Formas cartesiana y currificada

- Una función está en forma cartesiana si su argumento es una tupla. Por ejemplo,

```
suma_cartesiana :: (Int,Int) -> Int
suma_cartesiana (x,y)  = x+y
```

- En cambio, la función

```
suma_currificada :: Int -> Int -> Int
suma_currificada x y  = x+y
```

está en forma currificada.

Formas cartesiana y currificada

- `curry f` es la versión currificada de la función `f`. Por ejemplo,

```
curry suma_cartesiana 2 3 ~> 5
```

Prelude
```
curry :: ((a,b) -> c) -> (a -> b -> c)
curry f x y = f (x,y)
```

- `uncurry f` es la versión cartesiana de la función `f`. Por ejemplo,

```
uncurry suma_currificada (2,3) ~> 5
```

Prelude
```
uncurry     :: (a -> b -> c) -> ((a,b) -> c)
uncurry f p  = f (fst p) (snd p)
```

Tema 3: Estructuras de datos

1. Listas

2. Listas especiales

3. Tuplas

4. Tipos de datos definidos
   - Definiciones de tipos
   - Números racionales
   - Tipo de dato definido: Árboles
   - Árboles de búsqueda
   - Usos especiales de definiciones de datos

Definiciones de tipos con `type`

- Un punto es un par de números reales. Por ejemplo,

  ```
  (3.0,4.0) :: Punto
  ```

```haskell
type Punto = (Float, Float)
```

- `distancia_al_origen p` es la distancia del punto `p` al origen. Por ejemplo,

  ```
  distancia_al_origen (3,4) ⇝ 5.0
  ```

```haskell
distancia_al_origen :: Punto -> Float
distancia_al_origen (x,y) = sqrt (x*x+y*y)
```

Definiciones de tipos con `type`

- `distancia p1 p2` es la distancia entre los puntos `p1` y `p2`. Por ejemplo,

  ```
  distancia (2,4) (5,8) ⇝ 5.0
  ```

```haskell
distancia :: Punto -> Punto -> Float
distancia (x,y) (x',y') = sqrt((x-x')^2+(y-y')^2)
```

- Un camino es una lista de puntos. Por ejemplo,

  ```
  [(1,2),(4,6),(7,10)] :: Camino
  ```

```haskell
type Camino = [Punto]
```

Definiciones de tipos con `type`

- `longitud_camino c` es la longitud del camino c. Por ejemplo,

  `longitud_camino [(1,2),(4,6),(7,10)]` $\leadsto$ `10.0`

- Definición recursiva:

```
longitud_camino :: Camino -> Float
longitud_camino [_] = 0
longitud_camino (p:q:xs) =
    (distancia p q) + (longitud_camino (q:xs))
```

Definiciones de tipos con type

- Definición no recursiva:

```
longitud_camino' :: Camino -> Float
longitud_camino' xs =
    sum [distancia p q |
         (p,q) <- zip (init xs) (tail xs)]
```

Evaluación paso a paso:

```
longitud_camino [(1,2),(4,6),(7,10)]
= sum [distancia p q | (p,q) <- zip (init [(1,2),(4,6),(7,10)])
                                    (tail [(1,2),(4,6),(7,10)])]
= sum [distancia p q | (p,q) <- zip [(1,2),(4,6)] [(4,6),(7,10)]]
= sum [distancia p q | (p,q) <- [((1,2),(4,6)),((4,6),(7,10))]]
= sum [5.0,5.0]
= 10
```

Tema 3: Estructuras de datos

1. Listas

2. Listas especiales

3. Tuplas

4. Tipos de datos definidos
   - Definiciones de tipos
   - Números racionales
   - Tipo de dato definido: Árboles
   - Árboles de búsqueda
   - Usos especiales de definiciones de datos

Números racionales

- En esta sección se presentan los números racionales como una aplicación de tuplas y definición de tipos.
- Un número racional es un par de enteros

```haskell
type Racional = (Int, Int)
```

- Ejemplos de números racionales:

```haskell
qCero   = (0,1)
qUno    = (1,1)
qDos    = (2,1)
qTres   = (3,1)
qMedio  = (1,2)
qTercio = (1,3)
qCuarto = (1,4)
```

Números racionales

- `simplificar x` es el número racional `x` simplificado. Por ejemplo,

```
simplificar (12,24)    ⇝ (1,2)
simplificar (12,-24)   ⇝ (-1,2)
simplificar (-12,-24)  ⇝ (1,2)
simplificar (-12,24)   ⇝ (-1,2)
```

```
simplificar (n,d) =
    (((signum d)*n) `div` m, (abs d) `div` m)
    where m = gcd n d
```

Números racionales

- `gcd x y` es el máximo común divisor de `x` e `y`. Por ejemplo,

  ```
  gcd 6 15 ⇝ 3
  ```

Prelude

```
gcd 0 0  = error "Prelude.gcd: gcd 0 0 is undefined"
gcd x y  = gcd' (abs x) (abs y)
           where gcd' x 0 = x
                 gcd' x y = gcd' y (x `rem` y)
```

- Definición alternativa

```
gcd_alt x y = last (filter (divisible y') (divisores x'))
              where x'             = abs x
                    y'             = abs y
                    divisores x    = filter (divisible x)
                    divisible x y  = x `rem` y == 0
```

Números racionales

- Operaciones con números racionales. Por ejemplo,

```
qMul (1,2) (2,3) ⇝ (1,3)
qDiv (1,2) (1,4) ⇝ (2,1)
qSum (1,2) (3,4) ⇝ (5,4)
qRes (1,2) (3,4) ⇝ (-1,4)
```

```haskell
qMul, qDiv, qSum, qRes :: Racional -> Racional -> Racional
qMul (x1,y1) (x2,y2) = simplificar (x1*x2, y1*y2)
qDiv (x1,y1) (x2,y2) = simplificar (x1*y2, y1*x2)
qSum (x1,y1) (x2,y2) = simplificar (x1*y2+y1*x2, y1*y2)
qRes (x1,y1) (x2,y2) = simplificar (x1*y2-y1*x2, y1*y2)
```

Números racionales

- `escribeRacional x` es la cadena correspodiente al número racional `x`. Por ejemplo,

```
escribeRacional (10,12)               ~> "5/6"
escribeRacional (12,12)               ~> "1"
escribeRacional (qMul (1,2) (2,3))    ~> "1/3"
```

```
escribeRacional :: Racional -> String
escribeRacional (x,y)
    | y' == 1    = show x'
    | otherwise = show x' ++ "/" ++ show y'
    where (x',y') = simplificar (x,y)
```

Tema 3: Estructuras de datos

1. Listas

2. Listas especiales

3. Tuplas

4. Tipos de datos definidos
   - Definiciones de tipos
   - Números racionales
   - Tipo de dato definido: Árboles
   - Árboles de búsqueda
   - Usos especiales de definiciones de datos

Tipo de dato definido: Árboles

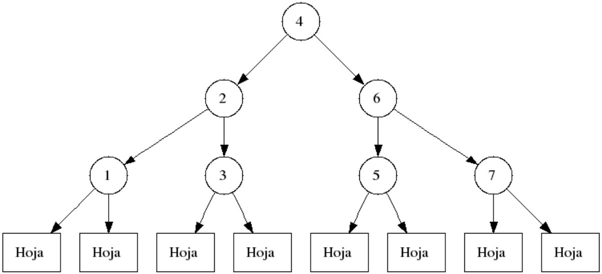
- Un árbol de tipo a es una hoja de tipo a o es un nodo de tipo a con dos hijos que son árboles de tipo a.

```
data Árbol a = Hoja
             | Nodo a (Árbol a) (Árbol a)
               deriving Show
```

  - El nombre del tipo de datos es `Árbol`
  - Las funciones constructoras son `Hoja` y `Nodo`
- El nombre del tipo de dato y de sus funciones constructoras tienen que empezar por mayúscula.

Tipo de dato definido: Árboles

- Ejemplo de árbol: El árbol de la figura



se representa por

```
ejÁrbol_1 = Nodo 4 (Nodo 2 (Nodo 1 Hoja Hoja)
                           (Nodo 3 Hoja Hoja))
                   (Nodo 6 (Nodo 5 Hoja Hoja)
                           (Nodo 7 Hoja Hoja))
```

Definición de funciones sobre árboles

- Las funciones sobre árboles se pueden definir mediante análisis de patrones con las funciones constructoras.
- `tamaño a` es el tamaño del árbol a; es decir, el número de nodos internos. Por ejemplo,

  ```
  tamaño ejÁrbol_1 ⇝ 7
  ```

```haskell
tamaño :: Árbol a -> Int
tamaño Hoja           = 0
tamaño (Nodo x a1 a2) = 1 + tamaño a1 + tamaño a2
```

Otras formas de árboles

- Árboles cuyos elementos no están en los nodos (como en `Árbol`), sino que están solamente en las hojas:

```
data Árbol2 a = Nodo2 (Árbol2 a) (Árbol2 a)
              | Fin2 a
```

- Árboles con información del tipo `a` en los nodos, e información del tipo `b` en las hojas:

```
data Árbol3 a b = Nodo3 a (Árbol3 a b) (Árbol3 a b)
                | Fin3 b
```

- Árboles que se dividen en tres en los nodos, y no en dos:

```
data Árbol4 a = Nodo4 a (Árbol4 a) (Árbol4 a) (Árbol4 a)
              | Fin4
```

Otras formas de árboles

- Árboles cuyo número de ramas bifurcadas de un nodo son variables:

```
data Árbol5 a = Nodo5 a [Árbol5 a]
```

- Árboles cuyos nodos solamente tienen una rama bifurcada:

```
data Árbol6 a = Nodo6 a (Árbol6 a)
              | Fin6
```

- Árboles con diferentes tipos de nodos:

```
data Árbol7 a b = Nodo7a Int a (Árbol7 a b) (Árbol7 a b)
                | Nodo7b Char (Árbol7 a b)
                | Fin7a b
                | Fin7b Int
```

Tema 3: Estructuras de datos

1. Listas

2. Listas especiales

3. Tuplas

4. Tipos de datos definidos
   - Definiciones de tipos
   - Números racionales
   - Tipo de dato definido: Árboles
   - Árboles de búsqueda
   - Usos especiales de definiciones de datos

Búsqueda en listas

- Búsqueda en lista: `elem e l` se verifica si `e` es un elemento de `l`.
- Búsqueda en lista ordenada: `elem_ord e l` se verifica si `e` es un elemento de la lista ordenada `l`. Por ejemplo,

```
elem_ord 3 [1,3,5] ⇝ True
elem_ord 2 [1,3,5] ⇝ False
```

```haskell
elem_ord :: Ord a => a -> [a] -> Bool
elem_ord _ []                       = False
elem_ord e (x:xs) | x < e           = elem_ord e xs
                  | x == e          = True
                  | otherwise       = False
```

Árbol de búsqueda

- Un árbol de búsqueda es un árbol binario en el que que todos los valores en el subárbol izquierdo son menores que el valor en el nodo mismo, y que todos los valores en el subárbol derecho son mayores. Por ejemplo, el `ejÁrbol_1` es un árbol de búsqueda.
- `elemÁrbol e x` se verifica si `e` es un elemento del árbol de búqueda `x`. Por ejemplo,

```
elemÁrbol 5 ejÁrbol_1 ⇝ True
elemÁrbol 9 ejÁrbol_1 ⇝ False
```

```
elemÁrbol :: Ord a => a -> Árbol a -> Bool
elemÁrbol e Hoja                        = False
elemÁrbol e (Nodo x izq der) | e==x = True
                             | e<x  = elemÁrbol e izq
                             | e>x  = elemÁrbol e der
```

Construcción de un árbol de búsqueda

- `insertaÁrbol e ab` inserta el elemento e en el árbol de búsqueda ab. Por ejemplo,

```
Main> insertaÁrbol 3 ejÁrbol_1
Nodo 4 (Nodo 2 (Nodo 1 Hoja Hoja)
               (Nodo 3 (Nodo 3 Hoja Hoja) Hoja))
       (Nodo 6 (Nodo 5 Hoja Hoja) (Nodo 7 Hoja Hoja))
```

```
insertaÁrbol :: Ord a => a -> Árbol a -> Árbol a
insertaÁrbol e Hoja = Nodo e Hoja Hoja
insertaÁrbol e (Nodo x izq der)
    | e <= x = Nodo x (insertaÁrbol e izq) der
    | e > x  = Nodo x izq (insertaÁrbol e der)
```

Construcción de un árbol de búsqueda

- `listaÁrbol l` es el árbol de búsqueda obtenido a partir de la lista `l`. Por ejemplo,

```
Main> listaÁrbol [3,2,4,1]
Nodo 1
     Hoja
     (Nodo 4
           (Nodo 2
                 Hoja
                 (Nodo 3 Hoja Hoja))
           Hoja)
```

```
listaÁrbol :: Ord a => [a] -> Árbol a
listaÁrbol = foldr insertaÁrbol Hoja
```

Ordenación mediante árboles de búsqueda

- `aplana ab` es la lista obtenida aplanando el árbol `ab`. Por ejemplo,

  `aplana (listaÁrbol [3,2,4,1])` ⇝ `[1,2,3,4]`

```
aplana :: Árbol a -> [a]
aplana Hoja               = []
aplana (Nodo x izq der) =
    aplana izq ++ [x] ++ aplana der
```

- `ordenada_por_árbol l` es la lista `l` ordenada mediante árbol de búsqueda. Por ejemplo,

  `ordenada_por_árbol [1,4,3,7,2]` ⇝ `[1,2,3,4,7]`

```
ordenada_por_árbol :: Ord a => [a] -> [a]
ordenada_por_árbol = aplana . listaÁrbol
```

Tema 3: Estructuras de datos

1. Listas

2. Listas especiales

3. Tuplas

4. Tipos de datos definidos
   - Definiciones de tipos
   - Números racionales
   - Tipo de dato definido: Árboles
   - Árboles de búsqueda
   - Usos especiales de definiciones de datos

Tipos finitos

- Un tipo finito es un tipo que contiene exactamente tantos elementos como funciones constructoras.
- Ejemplo de tipo finito:

```
data Dirección = Norte | Sur | Este | Oeste
```

- `mueve d p` es el punto obtenido moviendo el punto `p` una unidad en la dirección `d`. Por ejemplo,

```
mueve Sur (mueve Este (4,7)) ~> (5,6)
```

```
mueve :: Dirección -> (Int,Int) -> (Int,Int)
mueve Norte (x,y) = (x,y+1)
mueve Sur   (x,y) = (x,y-1)
mueve Este  (x,y) = (x+1,y)
mueve Oeste (x,y) = (x-1,y)
```

Unión de tipos

- Un entero–o–carácter es un objeto de la forma `Ent x`, donde `x` es un entero, o `Car y`, donde `y` es un carácter.

```
data Ent_o_Car = Ent Int | Car Char deriving Show
```

```
Main> :set +t
Main> Ent 3
Ent 3 :: Ent_o_Car
Main> Car 'd'
Car 'd' :: Ent_o_Car
Main> [Ent 3, Car 'd', Car 'e', Ent 7]
[Ent 3,Car 'd',Car 'e',Ent 7] :: [Ent_o_Car]
Main> :type Ent
Ent :: Int -> Ent_o_Car
Main> :type Car
Car :: Char -> Ent_o_Car
```

Tipos abstractos

- Un tipo abstracto consiste en una definición de datos y una serie de funciones que se pueden aplicar al tipo definido.
- Los racionales como tipo abstracto:

```
data Ratio =  Rac Int Int
```

- Los racionales se escriben en forma simplificada:

```
instance Show Ratio where
    show (Rac x 1) = show x
    show (Rac x y) = show x' ++ "/" ++ show y'
                     where (Rac x' y') = reduce (Rac x y)
```

Tipos abstractos

- Ejemplos de números racionales:

```
rCero   = Rac 0 1
rUno    = Rac 1 1
rDos    = Rac 2 1
rTres   = Rac 3 1
rMedio  = Rac 1 2
rTercio = Rac 1 3
rCuarto = Rac 1 4
```

```
Main> :set +t
Main> rDos
2 :: Ratio
Main> rTercio
1/3 :: Ratio
```

Tipos abstractos

- `reduce x` es el número racional `x` simplificado. Por ejemplo,

```
reduce (Rac  12  24) ⇝  1/2
reduce (Rac  12 -24) ⇝ -1/2
reduce (Rac -12 -24) ⇝  1/2
reduce (Rac -12  24) ⇝ -1/2
```

```haskell
reduce :: Ratio -> Ratio
reduce (Rac n d) =
    Rac (((signum d)*n) `div` m) ((abs d) `div` m)
    where m = gcd n d
```

Tipos abstractos

- Operaciones con números racionales. Por ejemplo,

```
rMul (Rac 1 2) (Rac 2 3) ⇝  1/3
rDiv (Rac 1 2) (Rac 1 4) ⇝  2
rSum (Rac 1 2) (Rac 3 4) ⇝  5/4
rRes (Rac 1 2) (Rac 3 4) ⇝ -1/4
```

```
rMul, rDiv, rSum, rRes :: Ratio -> Ratio -> Ratio
rMul (Rac a b) (Rac c d) = reduce (Rac (a*c) (b*d))
rDiv (Rac a b) (Rac c d) = reduce (Rac (a*d) (b*c))
rSum (Rac a b) (Rac c d) = reduce (Rac (a*d+b*c) (b*d))
rRes (Rac a b) (Rac c d) = reduce (Rac (a*d-b*c) (b*d))
```

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*.