

Tema 4: Definición de funciones

Informática (2017–18)

José A. Alonso Jiménez

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

Tema 4: Definición de funciones

1. Definiciones por composición
2. Definiciones con condicionales
3. Definiciones con ecuaciones con guardas
4. Definiciones con equiparación de patrones
 - Constantes como patrones
 - Variables como patrones
 - Tuplas como patrones
 - Listas como patrones
5. Expresiones lambda
6. Secciones

Definiciones por composición

- ▶ Decidir si un carácter es un dígito:

```
_____ Prelude _____  
isDigit :: Char -> Bool  
isDigit c = c >= '0' && c <= '9'
```

- ▶ Decidir si un entero es par:

```
_____ Prelude _____  
even :: (Integral a) => a -> Bool  
even n = n `rem` 2 == 0
```

- ▶ Dividir una lista en su n -ésimo elemento:

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

Definiciones por composición

- ▶ Decidir si un carácter es un dígito:

```
_____ Prelude _____  
isDigit :: Char -> Bool  
isDigit c = c >= '0' && c <= '9'
```

- ▶ Decidir si un entero es par:

```
_____ Prelude _____  
even :: (Integral a) => a -> Bool  
even n = n `rem` 2 == 0
```

- ▶ Dividir una lista en su n -ésimo elemento:

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

Definiciones por composición

- ▶ Decidir si un carácter es un dígito:

```
_____ Prelude _____  
isDigit :: Char -> Bool  
isDigit c = c >= '0' && c <= '9'
```

- ▶ Decidir si un entero es par:

```
_____ Prelude _____  
even :: (Integral a) => a -> Bool  
even n = n `rem` 2 == 0
```

- ▶ Dividir una lista en su n -ésimo elemento:

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

Definiciones por composición

- ▶ Decidir si un carácter es un dígito:

```
_____ Prelude _____  
isDigit :: Char -> Bool  
isDigit c = c >= '0' && c <= '9'
```

- ▶ Decidir si un entero es par:

```
_____ Prelude _____  
even :: (Integral a) => a -> Bool  
even n = n `rem` 2 == 0
```

- ▶ Dividir una lista en su n -ésimo elemento:

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

Definiciones con condicionales

- ▶ Calcular el valor absoluto (con condicionales):

```
_____ Prelude _____  
abs :: Int -> Int  
abs n = if n >= 0 then n else -n  
_____
```

- ▶ Calcular el signo de un número (con condicionales anidados):

```
_____ Prelude _____  
signum :: Int -> Int  
signum n = if n < 0 then (-1) else  
           if n == 0 then 0 else 1  
_____
```

Definiciones con condicionales

- ▶ Calcular el valor absoluto (con condicionales):

```
_____ Prelude _____  
abs :: Int -> Int  
abs n = if n >= 0 then n else -n  
_____
```

- ▶ Calcular el signo de un número (con condicionales anidados):

```
_____ Prelude _____  
signum :: Int -> Int  
signum n = if n < 0 then (-1) else  
           if n == 0 then 0 else 1  
_____
```


Definiciones con condicionales

- ▶ Calcular el valor absoluto (con condicionales):

```
_____ Prelude _____  
abs :: Int -> Int  
abs n = if n >= 0 then n else -n  
_____
```

- ▶ Calcular el signo de un número (con condicionales anidados):

```
_____ Prelude _____  
signum :: Int -> Int  
signum n = if n < 0 then (-1) else  
           if n == 0 then 0 else 1  
_____
```

Definiciones con ecuaciones guardadas

- ▶ Calcular el valor absoluto (con ecuaciones guardadas):

```
_____ Prelude _____  
abs n | n >= 0    = n  
      | otherwise = -n  
_____
```

- ▶ Calcular el signo de un número (con ecuaciones guardadas):

```
_____ Prelude _____  
signum n | n < 0    = -1  
         | n == 0   = 0  
         | otherwise = 1  
_____
```

Definiciones con ecuaciones guardadas

- ▶ Calcular el valor absoluto (con ecuaciones guardadas):

```
_____ Prelude _____  
abs n | n >= 0    = n  
      | otherwise = -n  
_____
```

- ▶ Calcular el signo de un número (con ecuaciones guardadas):

```
_____ Prelude _____  
signum n | n < 0    = -1  
         | n == 0   = 0  
         | otherwise = 1  
_____
```

Definiciones con ecuaciones guardadas

- ▶ Calcular el valor absoluto (con ecuaciones guardadas):

```
_____ Prelude _____  
abs n | n >= 0    = n  
      | otherwise = -n  
_____
```

- ▶ Calcular el signo de un número (con ecuaciones guardadas):

```
_____ Prelude _____  
signum n | n < 0    = -1  
         | n == 0   = 0  
         | otherwise = 1  
_____
```

Tema 4: Definición de funciones

1. Definiciones por composición
2. Definiciones con condicionales
3. Definiciones con ecuaciones con guardas
4. Definiciones con equiparación de patrones
 - Constantes como patrones
 - Variables como patrones
 - Tuplas como patrones
 - Listas como patrones
5. Expresiones lambda

Definiciones con equiparación de patrones: Constantes

- ▶ Calcular la negación:

```
_____ Prelude _____  
not :: Bool -> Bool  
not True  = False  
not False = True
```

- ▶ Calcular la conjunción (con valores):

```
_____ Prelude _____  
(amp) :: Bool -> Bool -> Bool  
True  amp True  = True  
True  amp False = False  
False amp True  = False  
False amp False = False
```

Definiciones con equiparación de patrones: Constantes

- ▶ Calcular la negación:

```
_____ Prelude _____  
not :: Bool -> Bool  
not True  = False  
not False = True
```

- ▶ Calcular la conjunción (con valores):

```
_____ Prelude _____  
(amp) :: Bool -> Bool -> Bool  
True  amp True  = True  
True  amp False = False  
False amp True  = False  
False amp False = False
```

Definiciones con equiparación de patrones: Constantes

- ▶ Calcular la negación:

```
_____ Prelude _____  
not :: Bool -> Bool  
not True  = False  
not False = True
```

- ▶ Calcular la conjunción (con valores):

```
_____ Prelude _____  
(amp) :: Bool -> Bool -> Bool  
True  amp True  = True  
True  amp False = False  
False amp True  = False  
False amp False = False
```

Tema 4: Definición de funciones

1. Definiciones por composición
2. Definiciones con condicionales
3. Definiciones con ecuaciones con guardas
4. Definiciones con equiparación de patrones
 - Constantes como patrones
 - Variables como patrones**
 - Tuplas como patrones
 - Listas como patrones
5. Expresiones lambda

Definiciones con equiparación de patrones: Variables

- ▶ Calcular la conjunción (con variables anónimas):

```
----- Prelude -----  
(amp) :: Bool -> Bool -> Bool  
True  amp True = True  
_     amp _    = False
```

- ▶ Calcular la conjunción (con variables):

```
----- Prelude -----  
(amp) :: Bool -> Bool -> Bool  
True  amp x = x  
False amp _ = False
```

Definiciones con equiparación de patrones: Variables

- ▶ Calcular la conjunción (con variables anónimas):

```
_____ Prelude _____  
(&&) :: Bool -> Bool -> Bool  
True  && True = True  
_     && _    = False
```

- ▶ Calcular la conjunción (con variables):

```
_____ Prelude _____  
(&&) :: Bool -> Bool -> Bool  
True  && x = x  
False && _ = False
```

Definiciones con equiparación de patrones: Variables

- ▶ Calcular la conjunción (con variables anónimas):

```
_____ Prelude _____  
(&&) :: Bool -> Bool -> Bool  
True  && True = True  
_     && _    = False
```

- ▶ Calcular la conjunción (con variables):

```
_____ Prelude _____  
(&&) :: Bool -> Bool -> Bool  
True  && x = x  
False && _ = False
```

Tema 4: Definición de funciones

1. Definiciones por composición
2. Definiciones con condicionales
3. Definiciones con ecuaciones con guardas
4. Definiciones con equiparación de patrones
 - Constantes como patrones
 - Variables como patrones
 - Tuplas como patrones**
 - Listas como patrones
5. Expresiones lambda

Definiciones con equiparación de patrones: Tuplas

- ▶ Calcular el primer elemento de un par:

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

- ▶ Calcular el segundo elemento de un par:

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

Definiciones con equiparación de patrones: Tuplas

- ▶ Calcular el primer elemento de un par:

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

- ▶ Calcular el segundo elemento de un par:

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

Definiciones con equiparación de patrones: Tuplas

- ▶ Calcular el primer elemento de un par:

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

- ▶ Calcular el segundo elemento de un par:

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

Tema 4: Definición de funciones

1. Definiciones por composición
2. Definiciones con condicionales
3. Definiciones con ecuaciones con guardas
4. Definiciones con equiparación de patrones
 - Constantes como patrones
 - Variables como patrones
 - Tuplas como patrones
 - Listas como patrones
5. Expresiones lambda

Definiciones con equiparación de patrones: Listas

- ▶ (test1 xs) se verifica si xs es una lista de 3 caracteres que empieza por 'a'.

```
test1 :: [Char] -> Bool
test1 ['a',_,_] = True
test1 _         = False
```

- ▶ Construcción de listas con (:)
| [1,2,3] = 1:[2,3] = 1:(2:[3]) = 1:(2:(3:[]))
- ▶ (test2 xs) se verifica si xs es una lista de caracteres que empieza por 'a'.

```
test2 :: [Char] -> Bool
test2 ('a':_) = True
test2 _       = False
```

Definiciones con equiparación de patrones: Listas

- ▶ (test1 xs) se verifica si xs es una lista de 3 caracteres que empieza por 'a'.

```
test1 :: [Char] -> Bool
test1 ['a',_,_] = True
test1 _          = False
```

- ▶ Construcción de listas con (:)
| [1,2,3] = 1:[2,3] = 1:(2:[3]) = 1:(2:(3:[]))
- ▶ (test2 xs) se verifica si xs es una lista de caracteres que empieza por 'a'.

```
test2 :: [Char] -> Bool
test2 ('a':_) = True
test2 _       = False
```

Definiciones con equiparación de patrones: Listas

- ▶ `(test1 xs)` se verifica si `xs` es una lista de 3 caracteres que empieza por `'a'`.

```
test1 :: [Char] -> Bool
test1 ['a',_,_] = True
test1 _         = False
```

- ▶ Construcción de listas con `(:)`
| `[1,2,3] = 1:[2,3] = 1:(2:[3]) = 1:(2:(3:[]))`
- ▶ `(test2 xs)` se verifica si `xs` es una lista de caracteres que empieza por `'a'`.

```
test2 :: [Char] -> Bool
test2 ('a':_) = True
test2 _       = False
```

Definiciones con equiparación de patrones: Listas

- ▶ Decidir si una lista es vacía:

```
----- Prelude -----  
null :: [a] -> Bool  
null []      = True  
null (_:_)  = False
```

- ▶ Primer elemento de una lista:

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

- ▶ Resto de una lista:

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

Definiciones con equiparación de patrones: Listas

- ▶ Decidir si una lista es vacía:

```
_____ Prelude _____  
null :: [a] -> Bool  
null []      = True  
null (_:_)  = False
```

- ▶ Primer elemento de una lista:

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

- ▶ Resto de una lista:

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

Definiciones con equiparación de patrones: Listas

- ▶ Decidir si una lista es vacía:

```
_____ Prelude _____  
null :: [a] -> Bool  
null []      = True  
null (_:_)  = False
```

- ▶ Primer elemento de una lista:

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

- ▶ Resto de una lista:

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

Definiciones con equiparación de patrones: Listas

- ▶ Decidir si una lista es vacía:

```
_____ Prelude _____  
null :: [a] -> Bool  
null []      = True  
null (_:_)  = False
```

- ▶ Primer elemento de una lista:

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

- ▶ Resto de una lista:

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

Expresiones lambda

- ▶ Las funciones pueden construirse sin nombrarlas mediante las expresiones lambda.
- ▶ Ejemplo de evaluación de expresiones lambda:

```
| Prelude> (\x -> x+x) 3  
| 6
```

Expresiones lambda y parcialización

Uso de las expresiones lambda para resaltar la parcialización:

- ▶ $(\text{suma } x \ y)$ es la suma de x e y .
- ▶ Definición sin lambda:

```
suma x y = x+y
```

- ▶ Definición con lambda:

```
suma' = \x -> (\y -> x+y)
```

Expresiones lambda y parcialización

Uso de las expresiones lambda para resaltar la parcialización:

- ▶ (suma x y) es la suma de x e y.
- ▶ Definición sin lambda:

```
suma x y = x+y
```

- ▶ Definición con lambda:

```
suma' = \x -> (\y -> x+y)
```

Expresiones lambda y parcialización

Uso de las expresiones lambda para resaltar la parcialización:

- ▶ (suma x y) es la suma de x e y.
- ▶ Definición sin lambda:

```
suma x y = x+y
```

- ▶ Definición con lambda:

```
suma' = \x -> (\y -> x+y)
```

Expresiones lambda y funciones como resultados

Uso de las expresiones lambda en funciones como resultados:

- ▶ `(const x y)` es `x`.
- ▶ Definición sin lambda:

```
----- Prelude -----  
const :: a -> b -> a  
const x = x
```

- ▶ Definición con lambda:

```
-----  
const' :: a -> (b -> a)  
const' x = \_ -> x  
-----
```

Expresiones lambda y funciones como resultados

Uso de las expresiones lambda en funciones como resultados:

- ▶ $(\text{const } x \ y)$ es x .
- ▶ Definición sin lambda:

```
Prelude
```

```
const :: a -> b -> a
const x = x
```

- ▶ Definición con lambda:

```
const' :: a -> (b -> a)
const' x = \_ -> x
```

Expresiones lambda y funciones como resultados

Uso de las expresiones lambda en funciones como resultados:

- ▶ $(\text{const } x \ y)$ es x .
- ▶ Definición sin lambda:

```
Prelude
```

```
const :: a -> b -> a
const x = x
```

- ▶ Definición con lambda:

```
const' :: a -> (b -> a)
const' x = \_ -> x
```

Expresiones lambda y funciones de sólo un uso

Uso de las expresiones lambda en funciones con sólo un uso:

- ▶ `(impares n)` es la lista de los `n` primeros números impares.
- ▶ Definición sin lambda:

```
impares n = map f [0..n-1]
  where f x = 2*x+1
```

- ▶ Definición con lambda:

```
impares' n = map (\x -> 2*x+1) [0..n-1]
```

Expresiones lambda y funciones de sólo un uso

Uso de las expresiones lambda en funciones con sólo un uso:

- ▶ `(impares n)` es la lista de los `n` primeros números impares.
- ▶ Definición sin lambda:

```
impares n = map f [0..n-1]
  where f x = 2*x+1
```

- ▶ Definición con lambda:

```
impares' n = map (\x -> 2*x+1) [0..n-1]
```

Expresiones lambda y funciones de sólo un uso

Uso de las expresiones lambda en funciones con sólo un uso:

- ▶ `(impares n)` es la lista de los `n` primeros números impares.
- ▶ Definición sin lambda:

```
impares n = map f [0..n-1]
  where f x = 2*x+1
```

- ▶ Definición con lambda:

```
impares' n = map (\x -> 2*x+1) [0..n-1]
```

Secciones

- ▶ Los **operadores** son las funciones que se escriben entre sus argumentos.
- ▶ Los operadores pueden convertirse en funciones prefijas escribiéndolos entre paréntesis.

- ▶ Ejemplo de conversión:

```
| Prelude> 2 + 3
```

```
5
```

```
| Prelude> (+) 2 3
```

```
5
```

- ▶ Ejemplos de secciones:

```
| Prelude> (2+) 3
```

```
5
```

```
| Prelude> (+3) 2
```

```
5
```

Expresión de secciones mediante lambdas

Sea $*$ un operador. Entonces

- ▶ $(*) = \lambda x \rightarrow (\lambda y \rightarrow x*y)$
- ▶ $(x*) = \lambda y \rightarrow x*y$
- ▶ $(*y) = \lambda x \rightarrow x*y$

Aplicaciones de secciones

- ▶ Uso en definiciones de funciones mediante secciones

```
suma'      = (+)
siguiente  = (1+)
inverso    = (1/)
doble      = (2*)
mitad      = (/2)
```

- ▶ Uso en signatura de operadores:

```
_____ Prelude _____
(&&) :: Bool -> Bool -> Bool
```

- ▶ Uso como argumento:

```
| Prelude> map (2*) [1..5]
| [2,4,6,8,10]
```

Bibliografía

1. R. Bird. *Introducción a la programación funcional con Haskell*. Prentice Hall, 2000.
 - ▶ Cap. 1: Conceptos fundamentales.
2. G. Hutton *Programming in Haskell*. Cambridge University Press, 2007.
 - ▶ Cap. 4: Defining functions.
3. B. O'Sullivan, D. Stewart y J. Goerzen *Real World Haskell*. O'Reilly, 2008.
 - ▶ Cap. 2: Types and Functions.
4. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. *Razonando con Haskell*. Thompson, 2004.
 - ▶ Cap. 2: Introducción a Haskell.
5. S. Thompson. *Haskell: The Craft of Functional Programming*, Second Edition. Addison-Wesley, 1999.
 - ▶ Cap. 3: Basic types and definitions.