

Programación Funcional

Tatiana Aparicio Yuja

Contenido del Curso

- Conceptos fundamentales
- Funciones
- Listas
- Recursión e inducción
- Tipos de datos definidos por el usuario
- Clases

Plan de Evaluación

- **Primer Parcial**
- **Segundo Parcil**
- **Examen Final**
- **Segunda Instancia**

Bibliografía

1. BIRD, Richard – WADLER, Philip. “Introduction to Functional Programming”. Prentice Hall International, 1988.
2. APARICIO Yuja, Nancy Tatiana. “Texto Didáctico sobre Tipos y Clases en Programación Funcional”.
3. INTERNET: <http://www.haskell.org>

The joy of functional programming (Benkart)

- La moda: No por que sea moda es la mejor opción
- Código comprensible, conciso
- Funciones matemáticas puras: No hay efectos colaterales, puedo combinar funciones para crear otras
- Programar : Que, no Como
- Permite usar núcleo de la PC: ganar eficiencia
- Ventana hacia el futuro
- Minimiza posibilidad de errores: Fácil detectar errores

Conceptos Fundamentales

- Paradigmas de Programación
 - Imperativo (como)
 - Declarativo (que) : Prog. Funcional

Programación Funcional

Programar = Modelar mediante funciones matemáticas un problema

Ejecutar un programa = Evaluar expresiones



Contexto
doble x= 2*x

```
miPrimerPrograma.hs: Bloque 1
Archivo Edición Formato Ver Ayuda
-- MI PRIMER PROGRAMA
doble x= 2*x
cuadrado x= x*x
primero (x,y)=x
```

```
*Main> :cd E:\Familia\Tatiana\materias\
Warning: changing directory causes all
because the search path has changed.
Prelude> :load "miPrimerPrograma.hs"
[1 of 1] Compiling Main
Ok, modules loaded: Main.
*Main> doble(2+3)
10
*Main> |
```

Ventajas PF

- Programas cortos y concisos
- Funciones son valores de primer orden:
 $f \ g = h$ / $f, g, y h$ son funciones
- Permite el Polimorfismo
 $id \ x = x$ $id :: a \rightarrow a$
- Manipulación formal de programas.
- Administración memoria no es tarea del programador
- No hay efectos colaterales

Ventajas PF

- ➤ Evaluación “Lazy”: manejar valores que todavía no han sido calculados, por ejemplo:

$$f\ g = p\ x$$

$$x = g\ h$$

$$h = x\ l$$

- ➤ Ideal para prototipo de compiladores

Desventajas

- Entrada/Salida

Evaluación/Reducción

Existen dos estrategias de evaluación:

Define el orden en que se evalúan las expresiones

LAZY

**Voy de afuera hacia
adentro**

**Sólo EVALÚO
ARGUMENTOS SI
LOS NECESITO**

EAGER

**Voy de adentro hacia
afuera**

**Primero EVALÚO
ARGUMENTOS**

Evaluación/Reducción

Estrategias de evaluación (ejemplo)

LAZY

cuadrado(doble 3)

$\Rightarrow (\text{doble } 3) * (\text{doble } 3)$

$\Rightarrow 6 * 6$

$\Rightarrow 36$

primero (2, 1+2*3*4)

$\Rightarrow 2$

EAGER

cuadrado(doble 3)

$\Rightarrow \text{cuadrado } 6$

$\Rightarrow 36$

primero (2, 1+2*3*4)

$\Rightarrow \text{primero } (2, 25)$

$\Rightarrow 5$

SCRIPT

doble x = 2*x

cuadrado x = x*x

primero (x,y) = x

Evaluación/Reducción

LAZY

primero (2,5/0)

=> 2



EAGER

primero(2,5/0)

=> error



SCRIPT

```
doble x= 2*x  
cuadrado x= x*x  
primero (x,y)=x
```

La evaluación Lazy garantiza que si existe un resultado, el compilador/intérprete puede calcularlo Ej primero(2,5/0).

Los compiladores/intérpretes funcionales puros siguen esta estrategia de evaluación.

En virtud a la evaluación lazy los lenguajes funcionales pueden manipular valores indefinidos.

Evaluación/Reducción

LAZY

primero (2,5/0)

=> 2



EAGER

primero(2,5/0)

=> error



```
WinGHCi
File Edit Actions Tools Help
[*Main> primero(2,5/0)
2
*Main> primero(2,7)
2
*Main> |
```

```
miPrimerPrograma.hs: Blo
Archivo Edición Formato Ver Ayuda
-- MI PRIMER PROGRAMA
doble x= 2*x
cuadrado x= x*x
primero (x,y)=x
```

```
Python 3.7
File Edit Shell Debug Options Window Help
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019) on win32
Type "help", "copyright", "credits" or "license()"
>>>
RESTART: C:\Users\Tatina\AppData\Local\Python\Python37\python.exe
hola
>>> primero(2,5/0)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    primero(2,5/0)
ZeroDivisionError: division by zero
>>> primero(2,7)
2
>>> |
```

```
*prueba.py - C
File Edit Format Run Options Window Help
doble=lambda x: 2*x
cuadrado=lambda x: x*x
primero=lambda x,y: x
sumaPar = lambda x,y: x+y
```

Vemos que
haskell (izq)
llega al
resultado,
python (der) no

Ejemplos en haskell

The background of the slide is a dark blue gradient. A thin, light blue curved line starts from the top left and sweeps across the upper half of the slide. In the bottom right corner, there is a lighter blue, semi-transparent curved shape that resembles a corner or a stylized 'L'.

Tipos Básicos en Hugs

Int	5
Float	5.0
Char	'a'
String	"hola"
Bool	True, False

Tipos Compuestos en Hugs

TIPOS

VALORES

Tuplas

(Int,Char)

(5,'e')

(Int, Bool, (Char,Int))

(5,True,('a',7))

Listas

[Char]

['h','o','l','a']="hola"

[String]

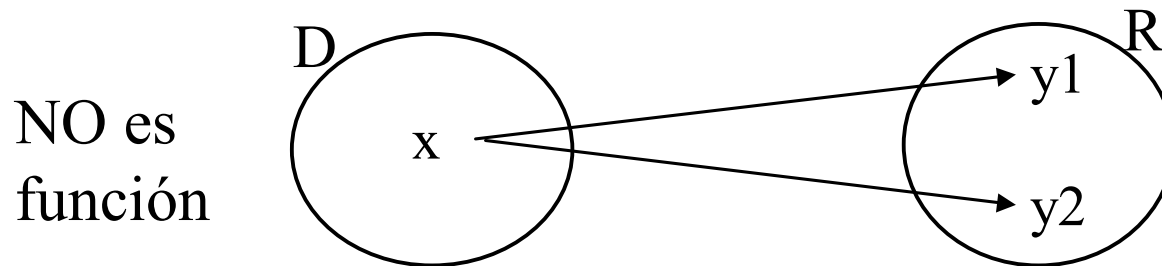
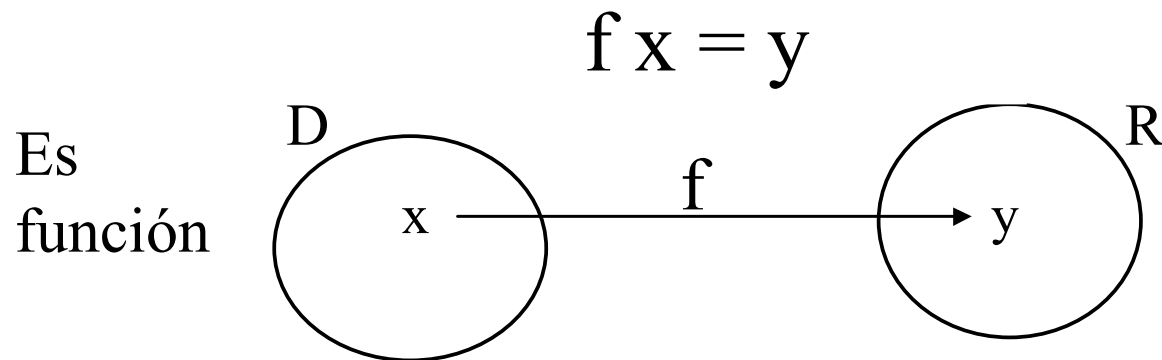
["hola","hello"]

[[Int]]

[[1,2,3],[5,6],[]]

Funciones

Una función f asocia cada x elemento de un Dominio (D) un único elemento y en el Rango (R)



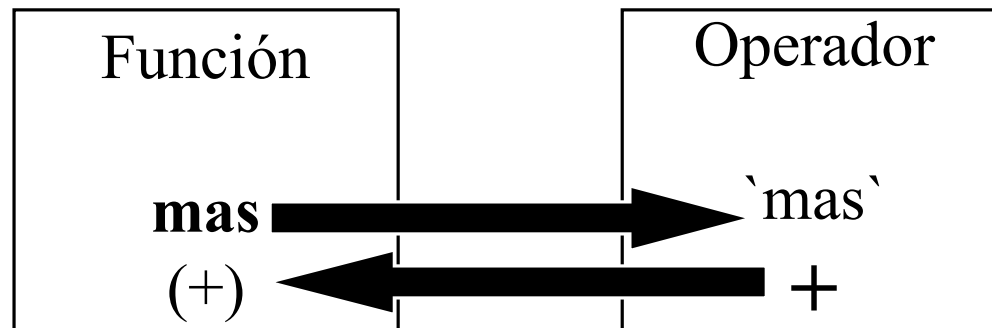
En el paradigma funcional se programa en base a funciones

Función = Operador

Para programar también se usan operadores, que son funciones con características particulares (van al medio de 2 parámetros).

En efecto, en haskell, cualquier operador puede ser usado como funcion y las funciones de dos parámetros pueden ser usadas como operadores

Sea: $\text{mas } x \ y = x + y$



? mas 2 4

6

? 2+4

6

? (+) 2 4

6

? 2 `mas` 4

6

? (6/2)

3

? (/2) 6

3

? (6/) 2

3

? (/) 6 2

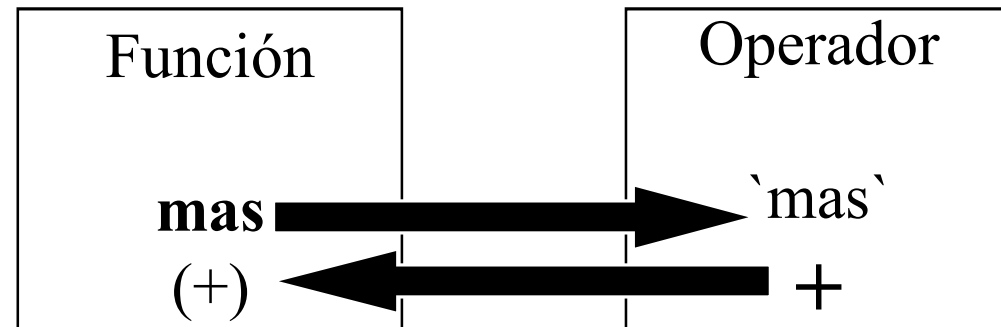
3

Función = Operador

Veamos en haskell

```
WinGHCi
File Edit Actions Tools Help
[Icons]
*Main> mas 2 3
5
*Main> 2 `mas` 3
5
*Main> 2+3
5
*Main> (+) 2 3
5
*Main> |
```

Sea: $\text{mas } x \ y = x+y$



```
? mas 2 4
6
? 2+4
6
? (+) 2 4
6
? 2 `mas` 4
6
```

```
? (6/2)
3
? (/2) 6
3
? (6/) 2
3
? (/) 6 2
3
```

Tipos de Funciones

- *Funciones de alto orden*
- *Funciones sobrecargadas*
- *Funciones Polimórficas*

Tipos de Funciones

- *Funciones Polimórficas*

Trabajan con cualquier tipo de datos

$\text{primero}(x,y) :: (a,b) \rightarrow a$

$\text{primero}(x,y) = x$

$\text{primero}(3,8) \Rightarrow 3$

$\text{primero}(\text{"mensaje"}, \text{True}) \Rightarrow \text{mensaje}$

$\text{primero}(+, *) \Rightarrow (+)$

$\text{primero}([1,2], [3,4,5]) \Rightarrow [1,2]$

Tipos de Funciones

- *Funciones de alto orden:*

aplicar:: $(a \rightarrow b) \rightarrow a \rightarrow b$

aplicar $f\ x = f\ x$

Reciben como argumento una función y/o devuelven como resultado una función

curry:: $((a,b) \rightarrow c) \rightarrow (a \rightarrow b \rightarrow c)$

curry $f = g$

where $g\ x\ y = f(x,y)$

componer:: $(b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$

componer $f\ g\ x = f\ (g\ x)$

Tipos de Funciones

Funciones sobrecargadas

Trabajan con un conjunto de tipos de datos

maximo::Ord a => a->a->a

maximo x y = if x > y then x else y

ordenado:: Ord a => (a->a->Bool)->[a] ->Bool


ordenado f [] = True

ordenado f [x] = True

ordenado f (x:y:xs) = (f x y) && ordenado f xs

Definición de Funciones

- Por Combinación
- Por Distinción de casos
- Expresiones if
- Expresiones case
- Definiciones locales
- Patrones
- Cálculo Lambda



```
cuad:: Int -> Int
cuad x=x*x
pote4:: Int -> Int
pote4 x = cuad (cuad x)
suma::Int -> Int -> Int
suma x y z= x+y+z
```

Combinamos funciones del contexto para crear nuevas funciones.
En el ejemplo definimos cuad en términos de *, y pote4 en términos de cuad

Definición de Funciones

- Por Combinación
- Por Distinción de casos
- Expresiones if
- Expresiones case
- Definiciones locales
- Patrones
- Cálculo Lambda

SINTAXIS


```
if exprLogica  
  then expr1  
  else expr2
```

Donde expr1 y expr2 deben ser del mismo tipo.

Se pueden anidar.

Definición de Funciones

- Por Combinación
- Por Distinción de casos
- Expresiones if
- Expresiones case
- Definiciones locales
- Patrones
- Cálculo Lambda



```
mayor:: Int -> Int -> Int
mayor x y = if x>y then x else y
may3:: Int -> Int -> Int ->Int
may3 x y z
  = if x > y then
      (if x > z then x else z)
      else (if y>z then y else z)
may3a:: Int -> Int -> Int ->Int
may3a x y z
  = if x > y && x>z then x
      else (if y>z then y else z)
```

Definición de Funciones

- Por Combinación
- Por Distinción de casos
- Expresiones if
- Expresiones case
- Definiciones locales
- Patrones
- Cálculo Lambda

SINTAXIS

f a1 a2 .. ak

| exprLog1 = expr1

| exprLog2 = expr2

.....

| otherwise = exprn

Donde expr1,expr2,..exprn son del mismo tipo

Devuelve la expr cuya exprLog es verdad

Deseables que las expresiones lógicas sean mutuamente excluyentes y exhaustivas

Definición de Funciones

- Por Combinación
- Por Distinción de casos
- Expresiones if
- Expresiones case
- Definiciones locales
- Patrones
- Cálculo Lambda

mayor:: Int -> Int -> Int

mayor x y | x > y = x
 | x < y = y

sgteVocal v

| v == 'a' = 'e'

| v == 'e' = 'i'


| v == 'i' = 'o'

| v == 'o' = 'u'

| v == 'u' = 'a'

| otherwise = '?'

Definición de Funciones

- Por Combinación
- Por Distinción de casos
- Expresiones if
- Expresiones case 
- Definiciones locales
- Patrones
- Cálculo Lambda

SINTAXIS

```
case expression of  
  expr1 -> exprRes1  
  expr2 -> exprRes2  
  ...  
  _ -> ExpResm
```

Donde $\text{expr1}, \text{expr2} \dots$ son del mismo tipo que expression

Y

$\text{exprRes1}, \text{exprRes2} \dots \text{expResm}$
son del mismo tipo


Definición de Funciones

- Por Combinación
- Por Distinción de casos
- Expresiones if
- Expresiones case →
- Definiciones locales
- Patrones
- Cálculo Lambda

```
case expression of  
  expr1 -> exprRes1  
  expr2 -> exprRes2  
  ...  
  _ -> ExpResn
```

```
num2bool:: Int -> Bool  
num2bool x  
  = case x of  
    1 -> True  
    0 -> False
```


Definición de Funciones

- Por Combinación
- Por Distinción de casos
- Expresiones if
- Expresiones case 
- Definiciones locales
- Patrones
- Cálculo Lambda

```
y:: Bool -> Bool -> Bool
y a b = case a of
    False -> False
    True  -> case b of
        False -> False
        _      -> True
```

```
menor x y::Int ->Int -> Int
menor x y= case x<y of
    True  = x
    _     = y
```


Definición de Funciones

- Por Combinación
- Por Distinción de casos
- Expresiones if
- Expresiones case
- Definiciones locales 
- Patrones
- Cálculo Lambda

SINTAXIS:

where:

f a1 a2 .. ak= expr

where defLocal1
defLocal2
.....

Expresión let...in:

f a1 a2 .. ak=

let

defLocal1
defLocal2
.....

in expr

Definición de Funciones

- Por Combinación
- Por Distinción de casos
- Expresiones if
- Expresiones case
- Definiciones locales →
- Patrones
- Cálculo Lambda

MAYOR DE 3 NÚMEROS:

where:

$m3 :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$

$m3\ x\ y\ z = \text{if } z > m \text{ then } z \text{ else } m$

where $m = \text{if } x > y \text{ then } x \text{ else } y$

let...in:

$m3 :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$

$m3\ x\ y\ z =$

let $m = \text{if } x > y \text{ then } x \text{ else } y$

in $\text{if } z > m \text{ then } z \text{ else } m$

Veamos la utilidad de las definiciones Locales

Definir una función que reciba como argumento 4 notas y devuelva el mensaje

Excelente si el prom está entre 90 y 100

Muy Bien si el prom está entre 80 y 89

Bien si el prom está entre 70 y 79

Regular si el prom está entre 51 y 69

Mal si el prom está entre 0 y 50

Notas inválidas en otro caso

Definiciones Locales



```
evalua:: Int -> Int -> Int -> Int -> String
```

```
evalua n1 n2 n3 n4
```

```
| (div (n1+n2+n3+n4) 4) >= 90 && (div (n1+n2+n3+n4) 4) <= 100 = "Excelent"
```

```
| (div (n1+n2+n3+n4) 4) >= 80 && (div (n1+n2+n3+n4) 4) <= 89 = "Muy Bien"
```

```
| (div (n1+n2+n3+n4) 4) >= 70 && (div (n1+n2+n3+n4) 4) <= 79 = "Bien"
```

```
| (div (n1+n2+n3+n4) 4) >= 51 && (div (n1+n2+n3+n4) 4) <= 69 = "Regular"
```

```
| (div (n1+n2+n3+n4) 4) >= 0 && (div (n1+n2+n3+n4) 4) <= 50 = "Mal"
```

```
| otherwise = "notas invalidas"
```

Definiciones Locales

```
evalua_v2:: Int -> Int -> Int -> Int -> String
```

```
evalua_v2 n1 n2 n3 n4
```

```
| prom>=90 && prom <=100 ="Excelente"
```

```
| prom>=80 && prom<=89 ="Muy Bien"
```

```
| prom>=70 && prom<=79 ="Bien"
```

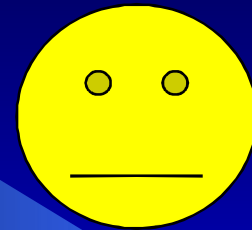
```
| prom>=51 && prom<=69 ="Regular"
```

```
| prom>=0 && prom <=50 ="Mal"
```

```
| otherwise ="notas invalidas"
```

```
where
```

```
    prom= div (n1+n2+n3+n4) 4
```



Definiciones Locales

```
evalua_v3:: Int -> Int -> Int -> Int -> String
```

```
evalua_v3 n1 n2 n3 n4
```

```
| promEn 90 100 ="Excelente"
```

```
| promEn 80 89 ="Muy Bien"
```

```
| promEn 70 79 ="Bien"
```

```
| promEn 51 69 ="Regular"
```

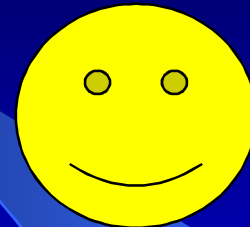
```
| promEn 0 50 ="Mal"
```

```
| otherwise ="notas invalidas"
```

```
where
```

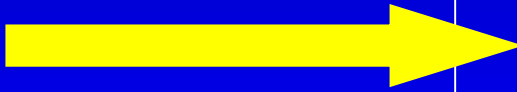
```
prom= div (n1+n2+n3+n4) 4
```

```
promEn li ls = prom>=li && prom <=ls
```



Las definiciones
locales permiten
factorizar código y
tener definiciones
más claras y
compactas

Definición de Funciones


- Por Combinación
- Por Distinción de casos
- Expresiones if
- Expresiones case
- Definiciones locales
- Patrones 
- Cálculo Lambda

Podemos usar patrones en los argumentos de una función para facilitar la definición.

El **patrón** es la **forma** que sigue el **argumento** de la función.

Ej. $\text{primero}(x,y) = x$

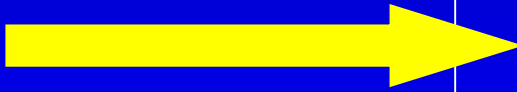
Definición de Funciones

- Por Combinación
- Por Distinción de casos
- Expresiones if
- Expresiones case
- Definiciones locales
- Patrones 
- Cálculo Lambda

Existen patrones:

- Constantes
- Estructurales
- Comodín
- @

Definición de Funciones


- Por Combinación
- Por Distinción de casos
- Expresiones if
- Expresiones case
- Definiciones locales
- Patrones 
- Cálculo Lambda

Patrón Constante

```
no :: Bool -> Bool
no True = False
no False = True

y :: Bool -> Bool -> Bool
y True True = True
y True False = False
y False True = False
y False False = False
```

Definición de Funciones

- Por Combinación
- Por Distinción de casos
- Expresiones if
- Expresiones case
- Definiciones locales
- Patrones 
- Cálculo Lambda

Patrón Constante

sgteVocal 'a' = 'e'

sgteVocal 'e' = 'i'

sgteVocal 'i' = 'o'

sgteVocal 'o' = 'u'

sgteVocal 'u' = 'a'

Definición de Funciones (Emparejamiento de patrones)

Patrón Constante

no :: Bool -> Bool

no **True** = False

no False = True

y :: Bool -> Bool -> Bool

y True True = True

y True False = False

y **False True** = False

y False False = False

? no **True**


False

? y **False True**

False

Toma la definición cuyo
parámetro formal
empareja con el
parámetro actual

Definición de Funciones

- Por Combinación
- Por Distinción de casos
- Expresiones if
- Expresiones case
- Definiciones locales
- Patrones 
- Cálculo Lambda

Patrón Comodín

no :: Bool -> Bool

no True = False

no _ = True

y :: Bool -> Bool -> Bool

y True True = True

y _ _ = False

El comodín (_)
empareja con todo.

Definición de Funciones (Patrones)

Patrón Estructural

primero :: (a,b) -> a

primero (x,y) = x

Función que devuelve la fecha mayor de dos

fecMay :: (Int,Int,Int) -> (Int,Int,Int) -> (Int,Int,Int)

fecMay (d1, m1, a1) (d2, m2, a2)

(a1 > a2) ((a1 == a2) && ((m1 > m2) (m1 == m2) && (d1 > d2)))	= (d1, m1, a1)
otherwise	= (d2, m2, a2)

Función que devuelve la longitud de una lista:

length :: [a] -> Int

length [] = 0

length (x:xs) = 1 + length xs

Patron @

Funcion que
recibe dos
fechas y
devuelve la
mayor

```
fecMay::(Int,Int,Int)->(Int,Int,Int)->(Int,Int,Int)
fecMay (d1, m1, a1) (d2, m2, a2)
  | a1>a2 = (d1,m1,a1)
  | a2>a1 = (d2,m2,a2)
  | m2>m1 = (d2,m2,a2)
  | m1>m2 = (d1,m1,a1)
  | d2>d1 = (d2,m2,a2)
  | otherwise = (d1,m1,a1)
```

Patron @

Funcion que
recibe dos
fechas y
devuelve la
mayor

El patrón @ sirve
para usar alias.
En el ejemplo, el
alias de la primera
fecha es f1 y de la
segunda fecha f2

```
fecMay::(Int,Int,Int)->(Int,Int,Int)->(Int,Int,Int)
fecMay (d1, m1, a1) (d2, m2, a2)
  | a1>a2 = (d1,m1,a1)
  | a2>a1 = (d2,m2,a2)
  | m2>m1 = (d2,m2,a2)
  | m1>m2 = (d1,m1,a1)
  | d2>d1 = (d2,m2,a2)
  | otherwise = (d1,m1,a1)
```

```
fecMay::(Int,Int,Int)->(Int,Int,Int)->(Int,Int,Int)
fecMay f1@(d1, m1, a1) f2@(d2, m2, a2)
  | a1>a2 = f1
  | a2>a1 = f2
  | m2>m1 = f2
  | m1>m2 = f1
  | d2>d1 = f2
  | otherwise = f1
```

Patron @

Funcion que recibe 4 fechas y devuelve la mayor

fecMay4v1 f1 f2 f3 f4
= fecMay (fecMay f1 f2) (fecMay f3 f4)



Sólo usar cuando es útil para la definición. En el ejemplo nos quedamos con la primera definición

fecMay4v2 f1@(d1, m1, a1) f2@(d2, m2, a2) f3@(d3, m3, a3) f4@(d4, m4, a4)
= fecMay (fecMay f1 f2) (fecMay f3 f4)



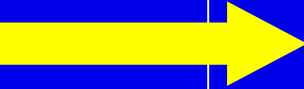
fecMay4v3 f1@(d1, m1, a1) f2@(d2, m2, a2) f3@(d3, m3, a3) f4@(d4, m4, a4)
= fecMay (fecMay (d1, m1, a1) (d2, m2, a2)) (fecMay (d3, m3, a3) (d4, m4, a4))



fecMay4v4 (d1, m1, a1) (d2, m2, a2) (d3, m3, a3) (d4, m4, a4)
= fecMay (fecMay (d1, m1, a1) (d2, m2, a2)) (fecMay (d3, m3, a3) (d4, m4, a4))



Definición de Funciones

- Por Combinación
 - Por Distinción de casos
 - Expresiones if
 - Expresiones case
 - Definiciones locales
 - Patrones
 - Cálculo Lambda
- 

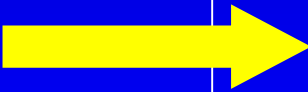
Expresiones Lambda

Podemos definir funciones usando el lenguaje de bajo nivel de la programación funcional (Cálculo Lambda)

En virtud al cálculo lambda en PF podemos manipular funciones como si fueran valores de primer orden ya que las funciones son expresiones

Permite usar funciones anónimas

Definición de Funciones

- Por Combinación
 - Por Distinción de casos
 - Expresiones if
 - Expresiones case
 - Definiciones locales
 - Patrones
 - Cálculo Lambda
- 

sintaxis

$f = \lambda a1 \rightarrow \lambda a2 \rightarrow \dots \lambda an$
 $\rightarrow \text{expresion}$

No permite usar: distinción de casos, where

Definición de Funciones

- Por Combinación
- Por Distinción de casos
- Expresiones if
- Expresiones case
- Definiciones locales
- Patrones
- Cálculo Lambda

Cálculo Lambda: $\lambda x \rightarrow 2 * x$

`doble :: Int -> Int`

`doble = (\x -> 2 * x)`

`suma :: Int -> Int -> Int`

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

Funciones anónimas:

? `(\x -> \y -> x * 2 + y) 3 4`
10

No permite usar: distinción de casos, where

Definición de Funciones

- Por Combinación
- Por Dist.de casos
- Expresiones if
- Expresiones case
- Definiciones locales
- Patrones
- Cálculo Lambda →

Mayor de 3 números:

```
may3 :: Int -> Int -> Int -> Int
```

```
may3 = (\x -> \y -> \z ->
```

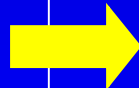
```
    let m = if x > y then x else y
```

```
    in  if z > m then z else m )
```

No permite usar: distinción de casos, where

Definición de Funciones

- Por Combinación
- Por Dist.de casos
- Expresiones if
- Expresiones case
- Definiciones locales
- Patrones
- Cálculo Lambda



Puedo usar patrones en definiciones lambda:

Definir una f. que reciba dos fechas y devuelva la que tiene el año mayor

anioMay= \f1@(_,_ ,a1) -> \f2@(_,_ ,a2)
 -> if a1>a2 then f1 else f2

Puedo usar definiciones locales con let-in

Ejemplo en haskell

```
miPrimerPrograma.hs: Bloc de notas
Archivo Edición Formato Ver Ayuda

-- MI PRIMER PROGRAMA EN
HASKELL
doble x= 2*x
cuadrado x= x*x
primero (x,y)=x
mas x y=x+y
unir= \x -> \y -> 10*x+y
```

Puedo usar expresiones lambda para definir funciones (ej. unir)

```
WinGHCi
File Edit Actions Tools Help
[Icons]

*Main> (\x -> 10*x) 3
30
*Main> primero ((\x -> 10*x), "hola") 3
30
*Main> unir 7 9
79
*Main> |
```

Puedo usar expresiones lambda como valores de primer orden, sin necesidad de darles un nombre