



Programació Funcional en Haskell

Jordi Petit

Departament de Ciències de la Computació
Universitat Politècnica de Catalunya

Programació Funcional en Haskell



Aperitiu

Jordi Petit

Departament de Ciències de la Computació
Universitat Politècnica de Catalunya

Haskell

Haskell és llenguatge de programació funcional pura.

No hi ha:

- assignacions,
- bucles,
- efectes laterals,
- gestió explícita de la memòria.

Hi ha:

- avaluació *lazy*,
- funcions com a objectes de primer ordre,
- sistema de tipus estàtic,
- inferència de tipus automàtica.

Haskell és elegant, concís i fa pensar d'una forma diferent!

Expressions

```
λ> 3 + 2 * 2  
👉 7
```

Expressions

```
λ> 3 + 2 * 2  
👉 7
```

```
λ> (3 + 2) * 2  
👉 10
```

Expressions

```
λ> 3 + 2 * 2  
👉 7
```

```
λ> (3 + 2) * 2  
👉 10
```

```
λ> even 42  
👉 True
```

Expressions

```
λ> 3 + 2 * 2  
👉 7
```

```
λ> (3 + 2) * 2  
👉 10
```

```
λ> even 42  
👉 True
```

```
λ> even(42)      -- 🤪 parèntesis absurds  
👉 True
```

Expressions

```
λ> 3 + 2 * 2  
👉 7
```

```
λ> (3 + 2) * 2  
👉 10
```

```
λ> even 42  
👉 True
```

```
λ> even(42) -- 🤪 parèntesis absurds  
👉 True
```

```
λ> even "Arnau" -- ❌ error de tipus
```


Expressions

```
λ> 3 + 2 * 2  
👉 7
```

```
λ> (3 + 2) * 2  
👉 10
```

```
λ> even 42  
👉 True
```

```
λ> even(42) -- 🤪 parèntesis absurds  
👉 True
```

```
λ> even "Arnau" -- ❌ error de tipus
```

```
λ> div 14 4  
👉 3
```

Tipus

```
λ> :type 'R'  
👉 'R' :: Char
```

Tipus

```
λ> :type 'R'  
👉 'R' :: Char
```

```
λ> :type "Marta"  
👉 "Marta" :: [Char]
```

Tipus

```
λ> :type 'R'  
👉 'R' :: Char
```

```
λ> :type "Marta"  
👉 "Marta" :: [Char]
```

```
λ> :type not  
👉 not :: Bool -> Bool
```

Tipus

```
λ> :type 'R'  
👉 'R' :: Char
```

```
λ> :type "Marta"  
👉 "Marta" :: [Char]
```

```
λ> :type not  
👉 not :: Bool -> Bool
```

```
λ> :type length  
👉 length :: [a] -> Int
```

Factorial

```
factorial :: Integer -> Integer
```

```
factorial 0 = 1
```

```
factorial n = n * factorial (n - 1)
```

Factorial

```
factorial :: Integer -> Integer
```

```
factorial 0 = 1
```

```
factorial n = n * factorial (n - 1)
```

```
λ> factorial 5
```



```
120
```

Factorial

```
factorial :: Integer -> Integer
```

```
factorial 0 = 1
```

```
factorial n = n * factorial (n - 1)
```

```
λ> factorial 5
```

```
👉 120
```

```
λ> map factorial [0..5]
```

```
👉 [1, 1, 2, 6, 24, 120]
```


Quicksort

```
quicksort [] = []  
quicksort (p:xs) = (quicksort menors) ++ [p] ++ (quicksort majors)  
  where  
    menors = [x | x <- xs, x < p]  
    majors = [x | x <- xs, x >= p]
```

Quicksort

```
quicksort [] = []  
quicksort (p:xs) = (quicksort menors) ++ [p] ++ (quicksort majors)  
  where  
    menors = [x | x <- xs, x < p]  
    majors = [x | x <- xs, x >= p]
```

```
λ> :type quicksort  
☞ quicksort :: Ord t => [t] -> [t]
```

Quicksort

```
quicksort [] = []  
quicksort (p:xs) = (quicksort menors) ++ [p] ++ (quicksort majors)  
  where  
    menors = [x | x <- xs, x < p]  
    majors = [x | x <- xs, x >= p]
```

```
λ> :type quicksort  
☞ quicksort :: Ord t => [t] -> [t]
```

```
λ> quicksort [5, 3, 6, 3, 1]  
☞ [1, 3, 3, 5, 6]
```

Quicksort

```
quicksort [] = []  
quicksort (p:xs) = (quicksort menors) ++ [p] ++ (quicksort majors)  
  where  
    menors = [x | x <- xs, x < p]  
    majors = [x | x <- xs, x >= p]
```

```
λ> :type quicksort  
☞ quicksort :: Ord t => [t] -> [t]
```

```
λ> quicksort [5, 3, 6, 3, 1]  
☞ [1, 3, 3, 5, 6]
```

```
λ> quicksort ["joan", "sara", "pep", "jana"]  
☞ ["jana", "joan", "pep", "sara"]
```

Arbres binaris

```
data Arbin t = Built
              | Node t (Arbin t) (Arbin t)
```

Arbres binaris

```
data Arbin t = Built
             | Node t (Arbin t) (Arbin t)
```

```
alcada :: Arbin t -> Integer
```

```
alcada Built = 0
alcada (Node x fe fd) = 1 + max (alcada fe) (alcada fd)
```

Arbres binaris

```
data Arbin t = Buit
             | Node t (Arbin t) (Arbin t)
```

```
alcada :: Arbin t -> Integer
```

```
alcada Buit = 0
alcada (Node x fe fd) = 1 + max (alcada fe) (alcada fd)
```

```
preordre :: Arbin t -> [t]
```

```
preordre Buit = []
preordre (Node x fe fd) = [x] ++ preordre fe ++ preordre fd
```

Sumari

- Haskell és llenguatge de programació funcional pura.
- Hem fet un primer tastet de les seves característiques.
- Hi ha moltes coses més!

Programació Funcional en Haskell



Eines

Jordi Petit

Departament de Ciències de la Computació
Universitat Politècnica de Catalunya

Eines necessàries

Glasgow Haskell Compiler (GHC):

- compilador (`ghc`)
- intèrpret (`ghci`)

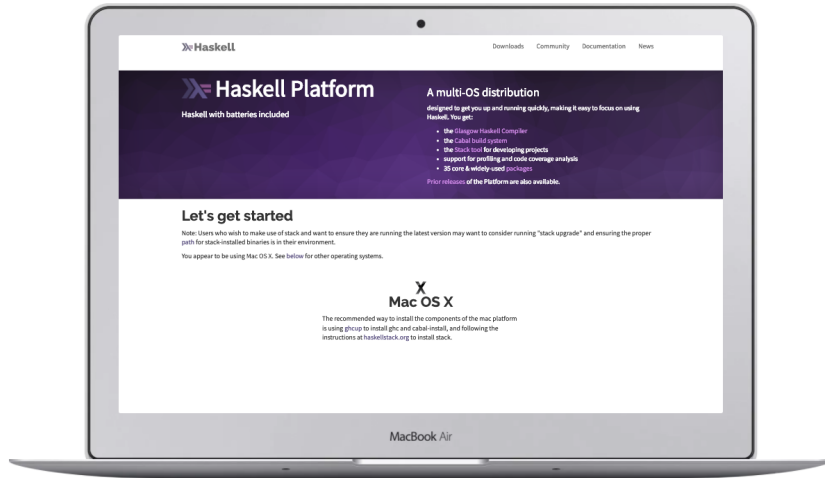
Editor de codi

Terminal

Jutge

Haskell Platform

<https://www.haskell.org/platform>



Instal·lació en Ubuntu



Instal·leu ghc amb apt.

```
sudo apt install ghc
```

Instal·lació en Mac



Useu [Homebrew](#)!

"El gestor de paquets per macOS que faltava"

```
brew install ghc
```

Instal·lació en Windows



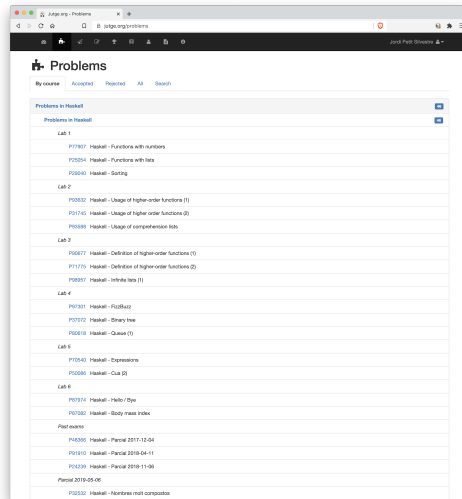
Useu [Chocolatey](#)!

"The Package Manager for Windows"

```
chocolatey install ghc
```



Apunteu-vos al curs "Problems in Haskell" de [Jutge.org](https://www.jutge.org).



Sumari

- Per treballar en Haskell només cal que instal·leu el GHC.
- És fàcil!
- GHC ofereix un compilador i un intèrpret.
- També necessitareu els vostres editors i emuladors de terminal favorits.
- Apunteu-vos al curs "Problemes en Haskell" de Jutge.org.

Programació Funcional en Haskell



Comencem!

Jordi Petit

Departament de Ciències de la Computació
Universitat Politècnica de Catalunya

Programació Funcional en Haskell



Tipus bàsics

Jordi Petit

Departament de Ciències de la Computació
Universitat Politècnica de Catalunya

Tipus bàsics

Booleans: `Bool`

Enters: `Int`, `Integer`

Reals: `Float`, `Double`

Caràcters: `Char`

Booleans

Tipus: Bool

Literals: False i True

Operacions:

```
not  :: Bool -> Bool      -- negació
(||)  :: Bool -> Bool -> Bool  -- disjunció
(&&)  :: Bool -> Bool -> Bool  -- conjunció
```

Exemples:

```
not True      ➡ False
not False     ➡ True

True || False ➡ True
True && False ➡ False

(False || True) and True ➡ True
not (not True)           ➡ True
not not True             ✗ -- vol dir: (not not) True
```

Enters

Tipus:

- `Int`: Enters de 64 bits en Ca2
- `Integer`: Enters (arbitràriament llargs)

Literals: `16`, `(-22)`, `587326354873452644428`

Operacions: `+`, `-`, `*`, `div`, `mod`, `rem`, `^`.

Operadors relacionals: `<`, `>`, `<=`, `>=`, `==`, `/=` (⚠ no `!=`)

Exemples:

```
3 + 4 * 5           ➡ 23
(3 + 4) * 5         ➡ 35
(3 + 4) * 5         ➡ 35
2^10                ➡ 1024
3 + 1 /= 4          ➡ False

div 11 2            ➡ 5
mod 11 2            ➡ 1
rem 11 2            ➡ 1
mod (-11) 2         ➡ 1
rem (-11) 2         ➡ -1
```

Reals

Tipus:

- `Float`: Reals de coma flotant de 32 bits
- `Double`: Reals de coma flotant de 64 bits

Literals: `3.14`, `1e-9`, `-3.0`

Operacions: `+`, `-`, `*`, `/`, `**`.

Operadors relacionals: `<`, `>`, `<=`, `>=`, `==`, `/=`

Conversió enter a real: `fromIntegral`

Conversió real a enter: `round`, `floor`, `ceiling`

Exemples:

```
10.0 / 3.0      ➡ 3.333333333333335
2.0 ** 3.0     ➡ 8.0
fromIntegral 4  ➡ 4.0
```

Caràcters

Tipus: `Char`

Literals: `'a'`, `'A'`, `'\n'`

Operadors relacionals: `<`, `>`, `<=`, `>=`, `==`, `/=`

Funcions de conversió: (cal un `import Data.Char`)

- `ord :: Char -> Int`
- `chr :: Int -> Char`

Precedència dels operadors

Precedència Associatius per l'esquerra No associatius Associatius per la dreta

9	!!	.
8		^, ^^, **
7	* / div mod rem quot	
6	+ -	
5		: ++
4		== /= < <= > >= elem notElem
3		&&
2		
1	>> >>=	
0		\$ \$! seq

Funcions predefinides habituals

és parell/senar:

```
even :: Integral a => a -> Bool
odd  :: Integral a => a -> Bool
```

mínim i màxim de dos valors:

```
min :: Ord a => a -> a -> a
max :: Ord a => a -> a -> a
```

màxim comú divisor, mínim comú multiple:

```
gcd :: Integral a => a -> a -> a
lcm :: Integral a => a -> a -> a
```

matemàtiques:

```
abs  :: Num a      => a -> a
sqrt :: Floating a => a -> a
log  :: Floating a => a -> a
exp  :: Floating a => a -> a
cos  :: Floating a => a -> a
```

Sumari

- Haskell ofereix tipus bàsics predefinits per:
 - booleans (`Bool`),
 - enters (`Int` i `Integer`),
 - reals (`Float` i `Double`), i
 - caràcters (`Char`).
- Cada tipus ofereix les operacions lògiques, aritmètiques i relacionals habituals a molts LPs.
- Compte: ⚠
 - Haskell es fa un embolic amb el canvi de signe: `(-22)`.
 - `mod` i `rem` que funcionen diferentment amb nombres negatius.
 - El \neq s'escriu `/=` i no `!=`.

Programació Funcional en Haskell



Funcions

Jordi Petit

Departament de Ciències de la Computació
Universitat Politècnica de Catalunya

Transparència referencial

- Les funcions en Haskell són *pures*: només retornen resultats calculats en relació als seus paràmetres.
- Les funcions no tenen efectes laterals (*side effects*).
 - no modifiquen els paràmetres
 - no modifiquen la memòria
 - no modifiquen l'entrada/sortida
- Una funció sempre retorna el mateix resultat aplicada sobre els mateixos paràmetres.

Definició de funcions

Els identificadors de funcions comencen amb minúscula.

Per introduir una funció:

1. Primer es dona la seva declaració de tipus (capçalera).
2. Després es dona la seva definició, utilitzant paràmetres formals.

Exemples:

```
doble :: Int -> Int           -- calcula el doble d'un valor
doble x = 2 * x

perimetre :: Int -> Int -> Int -- calcula l'àrea d'un rectangle
perimetre amplada alçada = doble (amplada + alçada)

xOr :: Bool -> Bool -> Bool   -- o exclusiva
xOr a b = (a || b) && not (a && b)

factorial :: Integer -> Integer -- calcula el factorial d'un natural
factorial n = if n == 0 then 1 else n * factorial (n - 1)
```

Definicions amb patrons

Les funcions es poden definir amb **patrons**:

```
factorial :: Integer -> Integer
-- calcula el factorial d'un natural

factorial 0 = 1
factorial n = n * factorial (n - 1)
```

L'avaluació dels patrons és de dalt a baix i retorna el resultat de la primera branca que casa.

Els patrons es consideren més elegants que el `if-then-else` i tenen moltes més aplicacions.

`_` representa una **variable anònima**: (no hi ha relació entre diferents `_`)

```
nand :: Bool -> Bool -> Bool          -- conjunció negada

nand True True = False
nand _ _ = True
```

Definicions amb guardes

Les funcions es poden definir amb **guardes**:

```
valAbs :: Int -> Int
-- retorna el valor absolut d'un enter

valAbs n
| n >= 0    = n
| otherwise = -n
```

L'avaluació de les guardes és de dalt a baix i retorna el resultat de la primera branca certa. (Error si cap és certa)

Les definicions per patrons també poden tenir guardes.

El `otherwise` és el mateix que `True`, però més llegible.

⚠ La igualtat va després de cada guarda!

Definicions locals

Per definir noms locals en una expressió s'utilitza el `let-in`:

```
fastExp :: Integer -> Integer -> Integer    -- exponenciació ràpida

fastExp _ 0 = 1
fastExp x n =
  let y = fastExp x n_halved
      n_halved = div n 2
  in
    if even n
    then y * y
    else y * y * x
```

El `where` permet definir noms en més d'una expressió:

```
fastExp :: Integer -> Integer    -- exponenciació ràpida

fastExp _ 0 = 1
fastExp x n
  | even n    = y * y
  | otherwise = y * y * x
  where
    y = fastExp x n_halved
    n_halved = div n 2
```

La identació del `where` defineix el seu àmbit.

Curricació

Totes les funcions tenen un únic paràmetre.

Les funcions de més d'un paràmetre retornen, en realitat, una nova funció.

No cal passar tots els paràmetres (aplicació parcial).

Curricació

Totes les funcions tenen un únic paràmetre.

Les funcions de més d'un paràmetre retornen, en realitat, una nova funció.

No cal passar tots els paràmetres (aplicació parcial).

Exemple:

`prod 3 5` és, en realitat, `(prod 3) 5`

Currficació

Totes les funcions tenen un únic paràmetre.

Les funcions de més d'un paràmetre retornen, en realitat, una nova funció.

No cal passar tots els paràmetres (aplicació parcial).

Exemple:

`prod 3 5` és, en realitat, `(prod 3) 5`

Primer apliquem 3 i el resultat és un funció que espera un altre enter.

```
prod :: Int -> Int -> Int
```

Currficació

Totes les funcions tenen un únic paràmetre.

Les funcions de més d'un paràmetre retornen, en realitat, una nova funció.

No cal passar tots els paràmetres (aplicació parcial).

Exemple:

`prod 3 5` és, en realitat, `(prod 3) 5`

Primer apliquem 3 i el resultat és un funció que espera un altre enter.

```
prod :: Int -> Int -> Int
```

```
prod :: Int -> (Int -> Int)
```

Currficació

Totes les funcions tenen un únic paràmetre.

Les funcions de més d'un paràmetre retornen, en realitat, una nova funció.

No cal passar tots els paràmetres (aplicació parcial).

Exemple:

`prod 3 5` és, en realitat, `(prod 3) 5`

Primer apliquem 3 i el resultat és un funció que espera un altre enter.

```
prod :: Int -> Int -> Int
```

```
prod :: Int -> (Int -> Int)
```

```
(prod 3) :: (Int -> Int)
```

Currficació

Totes les funcions tenen un únic paràmetre.

Les funcions de més d'un paràmetre retornen, en realitat, una nova funció.

No cal passar tots els paràmetres (aplicació parcial).

Exemple:

`prod 3 5` és, en realitat, `(prod 3) 5`

Primer apliquem 3 i el resultat és un funció que espera un altre enter.

```
prod :: Int -> Int -> Int
```

```
prod :: Int -> (Int -> Int)
```

```
(prod 3) :: (Int -> Int)
```

```
(prod 3) 5 :: Int
```

Inferència de tipus

Si no es dona la capçalera d'una funció, Haskell infereix el seu tipus.

Amb aquestes definicions,

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Haskell infereix que `factorial :: Num t => t -> t`.

Es pot preguntar el tipus d'una expressió amb `:type` a l'entèrpret:

```
λ> :type factorial
factorial :: Num t => t -> t
```

💬 Al principi, no useu la inferència de tipus (generalitza massa i perdeu disciplina).

💬 Pels problemes del Jutge, copieu les capçaleres donades als exercicis.

Notació prefixa/infixa

```
2 + 3      ➡ 5  
(+) 2 3    ➡ 5
```

Els operadors són infixes \Rightarrow posar-los entre parèntesis per fer-los prefixes

```
div 9 4     ➡ 2  
9 `div` 4   ➡ 2
```

Les funcions són prefixes \Rightarrow posar-les entre *backticks* per fer-les infixes

Sumari

- Les funcions en Haskell tenen un sol paràmetre (currificació).
 - `a -> b -> c` vol dir `a -> (b -> c)`.
 - `f x y` vol dir `(f x) y`.
- Per escriure una funció cal donar
 - la seva capçalera i
 - la seva definició.
- La inferència de tipus evita descriure les capçaleres de les funcions.
Eviteu-la al principi.
- Les definicions poden ser úniques o amb patrons i cada definició pot tenir guardes.
- Els patrons i les guardes es trien de dalt a baix.
- Es poden crear definicions locals amb el `let` i el `where` i es poden usar patrons localment amb el `case`.

Programació Funcional en Haskell



Tuples

Jordi Petit

Departament de Ciències de la Computació
Universitat Politècnica de Catalunya

Tuples

Una tupla és un tipus estructurat que permet desar diferents valors de tipus `t1`, `t2`, ..., `tn` en un únic valor de tipus `(t1, t2, ..., tn)`.

- El nombre de camps és fix.
- Els camps són de tipus heterogenis.

```
(3, 'z', False) :: (Int, Char, Bool)
(6, 9)         :: (Int, Int)
(True, (6, 9)) :: (Bool, (Int, Int))
```

```
caracterMesFrequent :: String -> (Char, Int)
caracterMesFrequent "PATATA"  👉 ('A', 3)
```

```
descomposicioHoraria :: Int -> (Int, Int, Int)    -- hores, minuts, segons

descomposicioHoraria segons = (h, m, s)
  where
    h = div segons 3600
    m = div (mod segons 3600) 60
    s = mod segons 60
```

Accés a tuples

Per a tuples de dos elements, es pot accedir amb `fst` i `snd`:

```
fst :: (a, b) -> a  
snd :: (a, b) -> b
```

```
fst (3, "rave")  
snd (3, "rave")
```



3

"rave"

Per a tuples generals, no hi ha definides funcions d'accés

⇒ Es poden crear fàcilment usant patrons:

```
primer (x, y, z) = x  
segon  (x, y, z) = y  
tercer (x, y, z) = z
```

```
primer (x, _, _) = x  
segon  (_, y, _) = y  
tercer (_, _, z) = z
```

Descomposició de tuples en patrons

Lleig:

```
distancia :: (Float, Float) -> (Float, Float) -> Float
-- calcula la distància entre dos punts 2D, cadascun donat amb una tupla

distancia p1 p2 = sqrt ((fst p1 - fst p2)^2 + (snd p1 - snd p2)^2)
```

Millor: Descompondre per patrons als propis paràmetres:

```
distancia (x1, y1) (x2, y2) = sqrt ((x1 - x2)^2 + (y1 - y2)^2)
```

També: Descompondre per patrons usant noms locals:

```
distancia p1 p2 = sqrt (sqr dx + sqr dy)
  where
    (x1, y1) = p1
    (x2, y2) = p2
    dx = x1 - x2
    dy = y1 - y2
    sqr x = x * x
```

Tupla buida (*unit*)

Existeix el tipus de tupla sense cap dada, que només té un possible valor: la dada buida.

Concepte semblant al `void` del C.

- Tipus: `()`
- Valor: `()`

En algun moment en farem ús.

Sumari

- Les tuples permeten agrupar diferents valors en un únic valor.
- El tipus `(t1, ..., tn)` representa el tipus tupla on el primer camp és de tipus `t1` i el darrer de tipus `tn`.
- Les funcions estàndard `fst` i `snd` permeten accedir al primer i al segon camp de les tuples de dos camps.
- Sovint és bo de descompondre en patrons.
- La tupla buida (*unit*) és el tipus `()` i només té un possible valor `()`.

Programació Funcional en Haskell



Llistes

Jordi Petit

Departament de Ciències de la Computació
Universitat Politècnica de Catalunya

Llistes

Una llista és un tipus estructurat que conté una seqüència d'elements, tots del mateix tipus.

`[t]` denota el tipus de les llistes d'elements de tipus `t`.

```
[ ]                -- llista buida
[3, 9, 27]         :: [Int]
[(1, "un"), (2, "dos"), (3, "tres")] :: [(Int, String)]
[[7], [3, 9, 27], [1, 5], []]      :: [[Int]]
[1 .. 10]          -- el mateix que [1,2,3,4,5,6,7,8,9,10]
[1, 3 .. 10]       -- el mateix que [1,3,5,7,9]
```

Constructors de llistes

Les llistes tenen dos **constructors**: `[]` i `:`

- La llista buida:

```
[] :: [a]
```

- Afegir per davant:

```
(:) :: a -> [a] -> [a]
```

Constructors de llistes

La notació

```
[16, 12, 21]
```

és una drecera per

```
16 : 12 : 21 : []
```

que vol dir

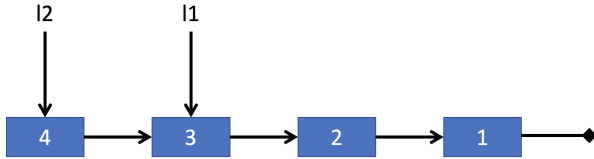
```
16 : (12 : (21 : []))
```

Implementació i eficiència

Les llistes de Haskell són llistes simplement encadenades.

Els constructors `[]` i `:` funcionen en temps constant (*DS sharing*).

```
l1 = 3 : 2 : 1 : []  
l2 = 4 : l1
```



L'operador `++` retorna la concatenació de dues llistes (temps proporcional a la llargada de la primera llista).

Llistes i patrons

La discriminació per patrons permet **descompondre** les llistes:

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

Diem que e_1 *matches* e_2 si existeix una substitució per les variables de e_1 que la fan igual que e_2 .

Exemples:

- $x:xs$ *matches* $[2, 5, 8]$ perquè $[2, 5, 8]$ és $2 : (5 : 8 : [])$ substituïnt x amb 2 i xs amb $(5 : 8 : [])$ que és $[5, 8]$.
- $x:xs$ *does not match* $[]$ perquè $[]$ i $:$ són constructors diferents.
- $x1:x2:xs$ *matches* $[2, 5, 8]$ substituïnt $x1$ amb 2 , $x2$ amb 5 i xs amb $[8]$.
- $x1:x2:xs$ *matches* $[2, 5]$ substituïnt $x1$ amb 2 , $x2$ amb 5 i xs amb $[]$.

Nota: El mecanisme de *matching* no és el mateix que el d'*unificació* (Prolog).

Llistes i patrons

La descomposició per patrons també es pot usar als `case`, `where` i `let`.

```
suma llista =  
  case llista of  
    []      -> 0  
    x:xs    -> x + suma xs
```

```
divImod n m  
  | n < m      = (0, n)  
  | otherwise  = (q + 1, r)  
  where (q, r) = divImod (n - m) m
```

```
primerIsegon llista =  
  let primer:segon:resta = llista  
  in (primer, segon)
```

Textos

Els textos (*strings*) en Haskell són llistes de caràcters.

El tipus `String` és una sinònim de `[Char]`.

Les cometes dobles són sucre sintàctic per definir textos.

```
nom1 :: [Char]
nom1 = 'p':'e':'p':[]
```

```
nom2 :: String
nom2 = "pepa"
```

```
λ> nom1 == nom2
```

```
👉 False
```

```
λ> nom1 < nom2
```

```
👉 True
```

Sumari

- Les llistes són seqüències d'elements del mateix tipus.
- `[a]` és el tipus de les llistes d'elements de tipus `a`.
- `String` és el tipus dels textos i correspon a `[Char]`.
- Les llistes tenen dos constructors:
 - la llista buida `[]`, i
 - l'operació d'afegir davant `(:)`.
- Per tant:
 - o bé una llista és buida,
 - o bé té un element seguit d'una subllista.
- La descomposició per patrons permet explotar-ho a l'hora d'escriure funcions.

Programació Funcional en Haskell



Funcions habituals per a llistes

Jordi Petit

Departament de Ciències de la Computació
Universitat Politècnica de Catalunya

head, last

- Signatura:

```
head :: [a] -> a
last :: [a] -> a
```

- Descripció:

- `head xs` és el primer element de la llista `xs`.
- `last xs` és el darrer element de la llista `xs`.

Error si `xs` és buida.

- Exemples:

```
λ> head [1..4]
1
λ> last [1..4]
4
```

tail, init

- Signatura:

```
tail :: [a] -> [a]
init  :: [a] -> [a]
```

- Descripció:

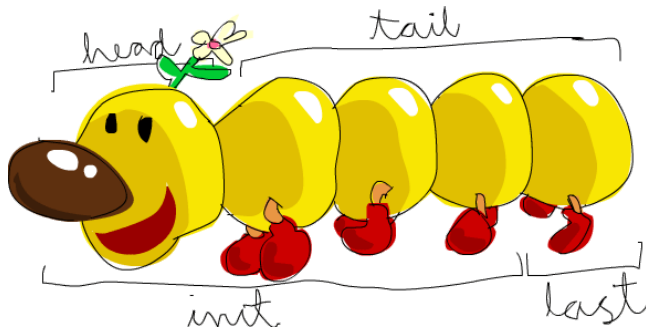
- `tail xs` és la llista `xs` sense el seu primer element.
- `init xs` és la llista `xs` sense el seu darrer element.

Error si `xs` és buida.

- Exemples:

```
λ> tail [1..4]
👉 [2, 3, 4]
λ> init [1..4]
👉 [1, 2, 3]
```

head, last, init, tail



Dibuix: [Learn You a Haskell, M. Lipovača](#)

reverse

- Signatura:

```
reverse :: [a] -> [a]
```

- Descripció:

`reverse xs` és la llista `xs` del revés.

- Exemples:

```
λ> reverse [1..4]  
👉 [4, 3, 2, 1]
```

length

- Signatura:

```
length :: [a] -> Int
```

- Descripció:

`length xs` és el nombre d'elements a la llista `xs`.

- Exemples:

```
λ> length []  
👉 0  
λ> length [1..5]  
👉 5  
λ> length "Marta"  
👉 5
```

null

- Signatura:

```
null :: [a] -> Bool
```

- Descripció:

`null xs` indica si la llista `xs` és buida.

- Exemples:

```
λ> null []  
👉 True  
λ> null [1..5]  
👉 False
```

elem

- Signatura:

```
elem :: Eq a => a -> [a] -> Bool
```

- Descripció:

`elem x xs` indica si `x` és a la llista `xs`.

- Exemples:

```
λ> elem 3 [1..10]
👉 True
λ> 3 `elem` [1..10]
👉 True
λ> 'k' `elem` "Jordi"
👉 False
```


Indexació: (!!)

- Signatura:

```
(!!) :: [a] -> Int -> a
```

- Descripció:

`xs !! i` és l'*i*-èsim element de la llista `xs` (començant per zero).

- Exemples:

```
λ> [1..10] !! 3
4
λ> [1..10] !! 11
Exception: index too large
```

Concatenació de dues llistes: (++)

- Signatura:

```
(++) :: [a] -> [a] -> [a]
```

- Descripció:

`xs ++ ys` és la llista resultant de posar `ys` darrera de `xs`.

- Exemples:

```
λ> "PEP" ++ "ET"  
👉 "PEPET"  
λ> [1..5] ++ [1..3]  
👉 [1,2,3,4,5,1,2,3]
```

maximum, minimum

- Signatura:

```
maximum :: Ord a => [a] -> a  
minimum :: Ord a => [a] -> a
```

- Descripció:

- `maximum xs` és l'element més gran de la llista (no buida!) `xs`.
- `minimum xs` és l'element més petit de la llista (no buida!) `xs`.

- Exemples:

```
λ> maximum [1..10]  
👉 10  
λ> minimum [1..10]  
👉 1  
λ> minimum []  
❌ Exception: empty list
```

sum, product

- Signatura:

```
sum    :: Num a => [a] -> a
product :: Num a => [a] -> a
```

- Descripció:

- `sum xs` és la suma de la llista `xs`.
- `prod xs` és el producte de la llista `xs`.

- Exemples:

```
λ> sum [1..5]
👉 15

factorial n = product [1 .. n]

λ> factorial 5
👉 120
```

and, or

- Signatura:

```
and :: [Bool] -> Bool  
or  :: [Bool] -> Bool
```

- Descripció:
 - `and bs` és la conjunció de la llista de booleans `bs`.
 - `or bs` és la disjunció de la llista de booleans `bs`.
- Observació:
 - Distingiu bé entre `and/or` i `(&&)/(||)`.

take, drop

- Signatura:

```
take :: Int -> [a] -> [a]
drop :: Int -> [a] -> [a]
```

- Descripció:

- `take n xs` és el prefixe de llargada `n` de la llista `xs`.
- `drop n xs` és el sufixe de la llista `xs` quan se li treuen els `n` primers elements.

- Exemples:

```
λ> take 3 [1 .. 7]
👉 [1, 2, 3]
λ> drop 3 [1 .. 7]
👉 [4, 5, 6, 7]
```

zip

- Signatura:

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

- Descripció:

`zip xs ys` és la llista que combina, en ordre, cada parell d'elements de `xs` i `ys`. Si en falten, es perden.

- Exemples:

```
λ> zip [1, 2, 3] ['a', 'b', 'c']  
👉 [(1, 'a'), (2, 'b'), (3, 'c')]  
λ> zip [1 .. 10] [1 .. 3]  
👉 [(1, 1), (2, 2), (3, 3)]
```

repeat

- Signatura:

```
repeat :: a -> [a]
```

- Descripció:

`repeat x` és la llista infinita on tots els elements són `x`.

- Exemples:

```
λ> repeat 3
☞ [3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, ...]
λ> take 4 (repeat 3)
☞ [3, 3, 3, 3]
```


concat

- Signatura:

```
concat :: [[a]] -> [a]
```

- Descripció:

`concat xs` és la llista que concatena totes les llistes de `xs`.

- Exemples:

```
λ> concat [[1, 2, 3], [], [3], [1, 2]]  
👉 [1, 2, 3, 3, 1, 2]
```


Sumari

- Existeixen moltes funcions predefinides sobre llistes que s'utilitzen habitualment.
- Exercici: implementeu vosaltres mateixos aquest funcions en termes dels constructors.
- Exemples:

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

```
myRepeat :: a -> [a]  
myRepeat x = x : myRepeat x
```

Exercicis Lab 1

1. Instal·leu-vos les eines per treballar.
2. Proveu de cercar documentació de funcions a [Hoogλe](#).
3. Feu aquests problemes de Jutge.org:
 - [P77907](#) Functions with numbers
 - [P25054](#) Functions with lists
 - [P29040](#) Sorting
 - Novetats:
 - Problemes amb puntuacions parcials 100. No cal que feu totes les funcions demanades.
 - Inspector de Haskell: comprova condicions de l'enunciat en el codi de la solució. Veredict NC  *Non compliant*. [TFG d'en Jan Mas]
4. Implementeu les funcions habituals sobre llistes vistes anteriorment.
 - Useu notació tipus `myLength` enlloc de `length` per evitar xocs de noms.
 - Useu recursivitat quan calgui o useu altres funcions `my*` que ja hagueu definit.

Programació Funcional en Haskell



Funcions d'ordre superior

Jordi Petit

Departament de Ciències de la Computació
Universitat Politècnica de Catalunya

Funcions d'ordre superior

Una **funció d'ordre superior** (FOS) és una funció que rep o retorna funcions.

Punt clau: les funcions són objectes de primera classe.

Funcions d'ordre superior

Una **funció d'ordre superior** (FOS) és una funció que rep o retorna funcions.

Punt clau: les funcions són objectes de primera classe.

Exemple en C++:

```
bool compare(int x, int y) {  
    return x > y;  
}  
  
int main() {  
    vector<int> v = { ... };  
    sort(v.begin(), v.end(), compare);    // sort és funció d'ordre superior  
}
```

Funcions d'ordre superior

Exemple: La funció predefinida `map` aplica una funció a cada element d'una llista.

```
map :: (a -> b) -> [a] -> [b]
```

```
map f [] = []
```

```
map f (x:xs) = f x : map f xs
```

```
λ> map odd [1..5]
```

```
👉 [True, False, True, False, True]
```

Funcions d'ordre superior

Exemple: La funció predefinida `(.)` retorna la composició de dues funcions:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
```

```
(f . g) x = f (g x)
```

```
λ> (reverse . sort) [5, 3, 5, 2]  
👉 [5, 5, 3, 2]
```


Funcions d'ordre superior

Exemple: La funció `apli2` aplica dos cops una funció a un element.

```
apli2 :: (a -> a) -> a -> a
```

```
apli2 f x = f (f x)
```

```
λ> apli2 sqrt 16.0  
👉 2.0
```

Funcions d'ordre superior

Exemple: La funció `apli2` aplica dos cops una funció a un element.

```
apli2 :: (a -> a) -> a -> a
```

```
apli2 f x = f (f x)
```

```
λ> apli2 sqrt 16.0  
👉 2.0
```

De forma equivalent:

```
apli2 :: (a -> a) -> (a -> a)
```

```
apli2 f = f . f
```

```
λ> apli2 sqrt 16.0  
👉 2.0
```

Funcions d'ordre superior

Exemple: La funció `apli2` aplica dos cops una funció a un element.

```
apli2 :: (a -> a) -> a -> a
apli2 f x = f (f x)
```

```
λ> apli2 sqrt 16.0
👉 2.0
```

De forma equivalent:

```
apli2 :: (a -> a) -> (a -> a)
apli2 f = f . f
```

```
λ> apli2 sqrt 16.0
👉 2.0
```

Petit exercici:

```
λ> per2 x = 2 * x
λ> apli2 (apli2 per2) 2
👉 ?
```

Funcions anònimes

Les funcions anònimes (funcions λ) són expressions que representen una funció sense nom.

```
\x -> x + 3      -- defineix funció anònima que, donada una x, retorna x + 3
                  -- si proveu d'escriure-la, Haskell s'enfada perquè no ho sap fer

(\x -> x + 3) 4    -- aplica la funció anònima sobre 4
👉 7
```

Funció amb nom:

```
doble x = 2 * x      -- equival a doble = \x -> 2 * x

λ> doble 3           👉 6

λ> map doble [1, 2, 3] 👉 [2, 4, 6]
```

Funció anònima:

```
λ> map (\x -> 2 * x) [1, 2, 3] 👉 [2, 4, 6]
```

Les funcions anònimes es solen usar quan són curtes i només s'utilitzen un cop.
També són útils per realitzar transformacions de programes.

Funcions anònimes

Múltiples paràmetres:

```
\x y -> x + y
```

és equivalent a

```
\x -> \y -> x + y
```

que vol dir

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

Seccions

Les **seccions** permeten aplicar operadors infixos parcialment.

Per la dreta:

```
( y) ≡ \x -> x    y
```

Per l'esquerra:

```
(y ) ≡ \x -> y    x
```

Exemples:

```
λ> doble = (* 2)           -- ≡ (2 *)
λ> doble 3
👉 6

λ> map (* 2) [1, 2, 3]      -- millor que map (\x -> x * 2) [1, 2, 3]
👉 [2, 4, 6]

λ> meitat = (/ 2)          -- ≠ (2 /)
λ> meitat 6
👉 3

λ> ésMajúscula = (`elem` ['A'..'Z'])
λ> ésMajúscula 'b'
👉 False
```

Sumari

- Una funció d'ordre superior és una funció que rep o retorna funcions.
- Les funcions anònimes permeten definir directament expressions que són funcions.
- Les seccions són aplicacions parcials d'operadors binaris.

Programació Funcional en Haskell



Funcions d'ordre superior habituals

Jordi Petit

Departament de Ciències de la Computació
Universitat Politècnica de Catalunya

Funcions d'ordre superior habituals

Algunes funcions d'ordre superior predefined s'utilitzen molt habitualment:

- `(.)`
- `($)`
- `const`
- `id`
- `flip`
- `map`
- `filter`
- `zipWith`
- `all`, `any`
- `dropWhile`, `takeWhile`
- `iterate`,
- `foldl`, `foldr`
- `scanl`, `scanr`

composició (.)

- Signatura:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

- Descripció:

$f \cdot g$ és la composició de les funcions f i g .

- Exemples:

```
λ> tresMesGrans = take 3 . reverse . sort
λ> :type tresMesGrans
tresMesGrans :: Ord a => [a] -> [a]
λ> tresMesGrans [3, 1, 2, 6, 7]
👉 [7, 6, 3]
```

aplicació (\$)

- Signatura:

```
( $\$$ ) :: (a -> b) -> a -> b
```

- Descripció:

$f \$ x$ és el mateix que $f x$. Sembla inútil, però degut a la baixa prioritat d'aquest operador, ens permet ometre molts parèntesis de tancar!

- Exemples:

```
λ> tail (tail (tail (tail "Jordi")))
👉 "i"
λ> tail $ tail $ tail $ tail "Jordi"
👉 "i"
```

const

- Signatura:

```
const :: a -> b -> a
```

- Descripció:

`const x` és una funció que sempre retorna `x`, independentment de què se li apliqui.

- Exemples:

```
λ> map (const 42) [1 .. 5]  
👉 [42, 42, 42, 42, 42]
```

id

- Signatura:

```
id :: a -> a
```

- Descripció:

`id` és la funció identitat. També sembla inútil, pero va bé en algun moment.

- Exemples:

```
λ> map id [1 .. 5]  
👉 [1, 2, 3, 4, 5]
```

flip

- Signatura:

```
flip :: (a -> b -> c) -> (b -> a -> c)
```

- Descripció:

`flip f` retorna la funció `f` però amb els seus dos paràmetres invertits. Es defineix per

```
flip f x y = f y x
```

- Exemples:

```
λ> meitat = flip div 2
```

```
λ> meitat 10
```

```
👉 5
```

flip

- Signatura:

```
flip :: (a -> b -> c) -> (b -> a -> c)
```

- Descripció:

`flip f` retorna la funció `f` però amb els seus dos paràmetres invertits. Es defineix per

```
flip f x y = f y x
```

- Exemples:

```
λ> meitat = flip div 2
```

```
λ> meitat 10
```

```
👉 5
```



map

- Signatura:

```
map :: (a -> b) -> [a] -> [b]
```

- Descripció:

`map f xs` és la llista que s'obté al aplicar la funció `f` a cada element de la llista `xs`, de forma que `map f [x1, x2, ..., xn]` és `[f x1, f x2, ..., f xn]`.

```
[y1, y2, y3, y4] = map f [x1, x2, x3, x4]
```

- Exemples:

```
λ> map even [2, 4, 6, 7] ➡ [True, True, True, False]
λ> map (*2) [2, 4, 6, 7] ➡ [4, 8, 12, 14]
```


filter

- Signatura:

```
filter :: (a -> Bool) -> [a] -> [a]
```

- Descripció:

`filter p xs` és la subllista dels elements de `xs` que compleixen el predicat `p`.

(Un **predicat** és una funció que retorna un Booleà.)

- Exemples:

```
λ> filter even [2, 1, 4, 6, 7]  
👉 [2, 4, 6]
```

zipWith

- Signatura:

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

- Descripció:

`zipWith op xs ys` és la llista obtinguda operant cada element de `xs` amb cada element de `ys` via la funció `op`, d'esquerra a dreta, mentre n'hi hagi.

```
[z1, z2, z3, z4] = zipWith f [x1, x2, x3, x4, x5] [y1, y2, y3, y4]
```

- Exemples:

```
λ> zipWith (+) [1, 2, 3] [5, 1, 8, 9]  
👉 [6, 3, 11]
```

all

- Signatura:

```
all :: (a -> Bool) -> [a] -> Bool
```

- Descripció:

`all p xs` indica si tots els elements de `xs` compleixen el predicat `p`.

- Exemples:

```
λ> all even [2, 1, 4, 6, 7]
👉 False
λ> all even [2, 4, 6]
👉 True
```

any

- Signatura:

```
any :: (a -> Bool) -> [a] -> Bool
```

- Descripció:

`all p xs` indica si algun dels elements de `xs` compleix el predicat `p`.

- Exemples:

```
λ> any even [2, 1, 4, 6, 7]
👉 True
λ> all odd [2, 4, 6]
👉 False
```

dropWhile

- Signatura:

```
dropWhile :: (a -> Bool) -> [a] -> [a]
```

- Descripció:

`dropWhile p xs` és la subllista de `xs` que elimina els primers elements de `xs` que compleixen el predicat `p` (fins al final o al primer que no la compleix).

- Exemples:

```
λ> dropWhile even [2, 4, 6, 7, 8]
👉 [7, 8]
λ> dropWhile even [2, 4]
👉 []
```

takeWhile

- Signatura:

```
takeWhile :: (a -> Bool) -> [a] -> [a]
```

- Descripció:

`takeWhile p xs` és la subllista de `xs` que conté els primers elements de `xs` que compleixen el predicat `p` (fins al final o al primer que no la compleix).

- Exemples:

```
λ> takeWhile even [2, 4, 6, 7, 8]  
👉 [2, 4, 6]  
λ> takeWhile even [1, 3]  
👉 []
```

iterate

- Signatura:

```
iterate :: (a -> a) -> a -> [a]
```

- Descripció:

`iterate f x` retorna la llista infinita `[x, f x, f (f x), f (f (f x)), ...]`.

```
ys = iterate f x
```

- Exemples:

```
λ> iterate (*2) 1  
👉 [1, 2, 4, 8, 16, ...]
```

foldl

- Signatura:

```
foldl :: (b -> a -> b) -> b -> [a] -> b
```

- Descripció:

`foldl ⊕ x0 xs` desplega un operador \oplus per l'esquerra, de forma que `foldl ⊕ x0 [x1, x2, ..., xn]` és $((x0 \oplus x1) \oplus x2) \oplus \dots \oplus xn$.

```
y = foldl f x0 [x1, x2, x3, x4]
```

- Exemples:

```
λ> foldl (+) 0 [3, 2, (-1)]      -- (((0 + 3) + 2) + (-1))  
👉 4
```


foldr

- Signatura:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

- Descripció:

`foldr ⊕ x0 xs` desplega un operador per la dreta, de forma que `foldr ⊕ x0 [x1, x2, ..., xn]` és `x1 ⊕ (x2 ... ⊕ (xn ⊕ x0))`.

```
y = foldr f x0 [x1, x2, x3, x4]
```

- Exemples:

```
λ> foldr (+) 0 [3, 2, (-1)]      -- 3 + ((2 + ((-1) + 0)))  
👉 4
```

scanl

- Signatura:

```
scanl :: (b -> a -> b) -> b -> [a] -> [b]
```

- Descripció:

`scanl f x0 xs` és com `foldl f x0 xs` però enlloc de retornar el valor final, retorna la llista amb tots els resultats intermigs.

```
[y0, y1, y2, y3, y4] = scanl f x0 [x1, x2, x3, x4]
```

- Exemples:

```
λ> scanl (+) 0 [3, 2, (-1)]  
👉 [0, 3, 5, 4]
```

scanr

- Signatura:

```
scanr :: (a -> b -> b) -> b -> [a] -> [b]
```

- Descripció:

`scanr f x0 xs` és com `foldr f x0 xs` però enlloc de retornar el valor final, retorna la llista amb tots els resultats intermigs.

```
[y0, y1, y2, y3, y4] = scanr f x0 [x1, x2, x3, x4]
```

- Exemples:

```
λ> scanr (+) 0 [3, 2, (-1)]  
👉 [4, 1, -1, 0]
```

Perspectiva

map

- C++

```
vector<X> xs = { ... };  
  
vector<Y> ys;  
for (int i = 0; i < xs.size(); ++i) {  
    ys.push_back(func(xs[i]));  
}
```

Perspectiva

map

- C++

```
vector<X> xs = { ... };  
  
vector<Y> ys;  
for (int i = 0; i < xs.size(); ++i) {  
    ys.push_back(func(xs[i]));  
}
```

- Haskell

```
ys = map func xs
```

Perspectiva

filter

- C++

```
vector<X> xs = { ... };  
  
vector<X> ys;  
for (int i = 0; i < xs.size(); ++i) {  
    if (pred(xs[i])) {  
        ys.push_back(xs[i]);  
    }  
}
```

Perspectiva

filter

- C++

```
vector<X> xs = { ... };  
  
vector<X> ys;  
for (int i = 0; i < xs.size(); ++i) {  
    if (pred(xs[i])) {  
        ys.push_back(xs[i]);  
    }  
}
```

- Haskell

```
ys = filter pred xs
```

Perspectiva

foldl

- C++

```
vector<X> xs = { ... };  
  
Y y = zero;  
for (int i = 0; i < xs.size(); ++i) {  
    y = oper(y, xs[i]);  
}
```


Perspectiva

foldl

- C++

```
vector<X> xs = { ... };  
  
Y y = zero;  
for (int i = 0; i < xs.size(); ++i) {  
    y = oper(y, xs[i]);  
}
```

- Haskell

```
y = foldl oper zero xs
```

Perspectiva

composició

- Haskell

```
(take 3 . reverse . sort) dades
```

Perspectiva

composició

- Haskell

```
(take 3 . reverse . sort) dades
```

- Shell

```
cat dades | sort | tac | head -3
```

Sumari

- Hem vist diverses funcions d'ordre superior molt habituals.
- Moltes d'aquestes funcions són abstraccions de recorreguts en LPs imperatius.
- Heu d'aprendre a identificar esquemes algorísmics habituals i ser capaços d'expressar-los amb funcions d'ordre superior.

Programació Funcional en Haskell



Aplicacions de funcions d'ordre superior

Jordi Petit

Departament de Ciències de la Computació
Universitat Politècnica de Catalunya

Diccionaris amb FOSs

Volem definir un Diccionari de Strings a Ints amb valors per defecte usant funcions d'ordre superior.

Diccionaris amb FOSs

Volem definir un Diccionari de Strings a Ints amb valors per defecte usant funcions d'ordre superior.

Interficie

```
type Dict = (String -> Int)      -- Defineix un tipus sinònim a la typedef

create :: Int -> Dict
search :: Dict -> String -> Int
insert :: Dict -> String -> Int -> Dict
```

Diccionaris amb FOSs

Volem definir un Diccionari de Strings a Ints amb valors per defecte usant funcions d'ordre superior.

Interficie

```
type Dict = (String -> Int)      -- Defineix un tipus sinònim a la typedef

create :: Int -> Dict
search :: Dict -> String -> Int
insert :: Dict -> String -> Int -> Dict
```

Primera versió

```
type Dict = (String -> Int)

create def = \key -> def

search dict key = dict key

insert dict key value = \x ->
  if key == x then value
  else search dict x
```


Diccionaris

Volem definir un TAD Diccionari de Strings a Ints amb valors per defecte usant funcions d'ordre superior.

Interficie

```
type Dict = (String -> Int)    -- Defineix un tipus sinònim a la typedef

create :: Int -> Dict
search :: Dict -> String -> Int
insert :: Dict -> String -> Int -> Dict
```

Primera versió

```
type Dict = (String -> Int)

create def = \key -> def

search dict key = dict key

insert dict key value = \x ->
  if key == x then value
  else search dict x
```

Segona versió

```
type Dict = (String -> Int)

create = const

search = ($)

insert dict key value x
  | key == x      = value
  | otherwise     = dict x
```

Dividir i vèncer

Funció d'ordre superior genèrica `dIv` per l'esquema de dividir i vèncer.

Interfície

```
dIv :: (a -> Bool) -> (a -> b) -> (a -> (a, a)) -> (a -> (a, a) -> (b, b) -> b) -> a -> b
```

on `a` és el tipus del problema, `b` és el tipus de la solució, i

`dIv` `trivial` `directe` `dividir` `vèncer` `x` utilitza:

- `trivial :: a -> Bool` per saber si un problema és trivial.
- `directe :: a -> b` per solucionar directament un problema trivial.
- `dividir :: a -> (a, a)` per dividir un problema no trivial en un parell de subproblemes més petits.
- `vèncer :: a -> (a, a) -> (b, b) -> b` per, donat un problema no trivial, els seus subproblemes i les seves respectives subsolucions, obtenir la solució al problema original.
- `x :: a` denota el problema a solucionar.

Dividir i vèncer

Solució

```
dIv :: (a -> Bool) -> (a -> b) -> (a -> (a, a)) -> (a -> (a, a) -> (b, b) -> b) -> a -> b

dIv trivial directe dividir vèncer x
| trivial x      = directe x
| otherwise      = vèncer x (x1, x2) (y1, y2)
                  where
                    (x1, x2) = dividir x
                    y1 = dIv trivial directe dividir vèncer x1
                    y2 = dIv trivial directe dividir vèncer x2
```

Dividir i vèncer

Solució capturant el context

```
dIv :: (a -> Bool) -> (a -> b) -> (a -> (a, a)) -> (a -> (a, a) -> (b, b) -> b) -> a -> b

dIv trivial directe dividir vèncer = dIv'
  where
    dIv' x
      | trivial x = directe x
      | otherwise = vèncer x (x1, x2) (y1, y2)
                          where
                            (x1, x2) = dividir x
                            y1 = dIv' x1
                            y2 = dIv' x2
```

Dividir i vèncer

Implementació de Quicksort amb Dividir i vèncer

```
qs :: Ord a => [a] -> [a]

qs = div trivial directe dividir vèncer
  where
    trivial [] = True
    trivial [_] = True
    trivial _ = False

    directe = id

    dividir (x:xs) = (menors, majors)
      where menors = filter (<= x) xs
            majors = filter (> x) xs

    dividir' (x:xs) = partition (<= x) xs      -- equivalent amb funció predefinida

    vèncer (x:_ ) _ (ys1, ys2) = ys1 ++ [x] ++ ys2
```

Dividir i vèncer

Implementació de Quicksort amb Dividir i vèncer

```
qs :: Ord a => [a] -> [a]

qs = dIv trivial directe dividir vèncer
  where
    trivial []      = True
    trivial [_]     = True
    trivial _       = False

    directe = id

    dividir (x:xs) = (menors, majors)
      where menors = filter (<= x) xs
            majors = filter (> x) xs

    dividir' (x:xs) = partition (<= x) xs      -- equivalent amb funció predefinida

    vèncer (x:_ ) _ (ys1, ys2) = ys1 ++ [x] ++ ys2
```

Exercicis:

- Escriviu ordenació per fusió amb `dIv`.
- Afegiu una FOS com a criteri de comparació per l'ordenació.
- Escriviu variant de dividir i vèncer amb nombre variable de subproblemes.

Sumari

- Hem vist com usar FOS per definir una estructura de dades:
 - les funcions són dades!
- Hem vist com usar FOS per definir un esquema algorísmic:
 - qualsevol millora a la nostra implementació millorarà totes les seves aplicacions.

Exercicis Lab 2

1. Feu aquests problemes de Jutge.org:

- [P93632](#) Usage of higher-order functions (1)
- [P31745](#) Usage of higher order functions (2)
- [P90677](#) Definition of higher-order functions (1)
- [P71775](#) Definition of higher-order functions (2)

2. Re-implementeu les funcions habituals sobre llistes.

- Useu `myLength` enlloc de `length` per evitar xocs de noms.
- No useu recursivitat: useu funcions d'ordre superior.

3. Busqueu a [Hoogʌe](#) informació sobre aquestes funcions:

- `foldl1`, `foldr1`, `scanl1`, `scanr1`
- `partition`
- `concatMap`
- `zipWith3`
- `mapAccumL`, `mapAccumR`

Programació Funcional en Haskell



Llistes per comprensió

Jordi Petit

Departament de Ciències de la Computació
Universitat Politècnica de Catalunya

Llistes amb rangs

Rangs

```
λ> [1 .. 10]
👉 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
λ> [10 .. 1]
👉 []
λ> ['E' .. 'J']
👉 ['E', 'F', 'G', 'H', 'I', 'J']
```

Rangs amb salt

```
λ> [10, 20 .. 100]
👉 [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
λ> [10, 9 .. 1]
👉 [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

λ> [1, 2, 4, 8, 16 .. 256]
❌ -- no fa miracles
```

Rangs sense final

```
λ> [1..]
👉 [1, 2, 3, 4, 5, 6, 7, 8, 9, ..... ]
λ> [1, 3 ..]
👉 [1, 3, 5, 7, 9, 11, 13, 15, ..... ]
```

Llistes per comprensió

Una **llista per comprensió** és una construcció per crear, filtrar i combinar llistes.

Sintàxi semblant a la notació matemàtica de construcció de conjunts.

Ternes pitagòriques en matemàtiques: $\{(x,y,z) \mid 0 < x \leq y \leq z, x^2 + y^2 = z^2\}$

Ternes pitagòriques en Haskell (fins a `n`):

```
λ> ternes n = [(x, y, z) | x <- [1..n], y <- [x..n], z <- [y..n], x*x + y*y == z*z]
-- gens eficient

λ> ternes 20
👉 [(3,4,5),(5,12,13),(6,8,10),(8,15,17),(9,12,15),(12,16,20)]
```

Llistes per comprensió

Ús bàsic: expressió amb generador (semblant a `map`)

```
[x*x | x <- [1..100]]
```

Filtre (semblant a `map` i `filter`)

```
[x*x | x <- [1..100], capicua x]
```

Múltiples filtres

```
[x | x <- [1..100], x `mod` 3 == 0, x `mod` 5 == 0]
```

Múltiples generadors (producte cartesià)

```
[(x, y) | x <- [1..10], y <- [1..10]]
```

Introducció de noms

```
[q | x <- [10..], let q = x*x, let s = show q, s == reverse s]
```

Llistes per comprensió


Compte amb l'ordre

```
[(x, y) | x <- [1..n], y <- [1..m], even x]  
[(x, y) | x <- [1..n], even x, y <- [1..m]]
```

Ternes pitagòriques

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

```
ternes n = [(x, y, z) | x <- [1..n],  
  y <- [x..n],  
  let z = floor $ sqrt $ fromIntegral $ x*x + y*y,  
  z <= n,  
  x*x + y*y == z*z]
```



Perspectiva

Haskell

```
[(x, y) | x <- xs, y <- ys, f x == g y, even x]
```

Python

```
[(x, y) for x in xs for y in ys if x.f == y.g and x%2 == 0]
```

SQL

```
SELECT *  
FROM xs  
JOIN ys  
WHERE xs.f = ys.g  
AND xs % 2 = 0
```

C++

```
list<pair<X, Y>> l;  
for (X x : xs)  
    for (Y y : ys)  
        if (x.f == y.g and x%2 == 0)  
            l.push_back({x, y});
```

Sumari

- Les llistes per comprensió de Haskell són una notació còmoda, semblant als conjunts per comprensió en matemàtiques.
- Les llistes per comprensió permeten crear noves llistes a partir d'altres llistes.
- Les llistes per comprensió tenen una funció de sortida, un o més generadors i cap o més predicats.

Programació Funcional en Haskell



Avaluació mandrosa

Jordi Petit

Departament de Ciències de la Computació
Universitat Politècnica de Catalunya

Avaluació mandrosa

- L'**avaluació mandrosa** (*lazy*) només avalua el que cal.
- Un *thunk* representa un valor que encara no ha estat avaluat.
- L'avaluació mandrosa no avalua els *thunks* fins que no ho necessita.
- Les expressions es tradueixen en un graf (no un arbre) que és recorregut per obtenir els elements necessaris.
- Això provoca cert indeterminisme en com s'executa.
- Ineficiència(?). Depen del compilador i depen del cas.
- Permet tractar estructures potencialment molt grans o "infinites".

Avaluació mandrosa: C++ vs Haskell

```
int f (int x, int y) { return x; }

int main() {
    int a, b;
    cin >> a >> b;
    cout << f(a, a / b);
}
```

💣: Divisió per zero quan b és zero.

```
int f (int x, int y) { return x; }
int h (int x)        { for (;;); }

int main() {
    int a, b;
    cin >> a >> b;
    cout << f(a, h(b));
}
```

💣: Es penja.

```
if (x != 0 ? 1 / x : 0) { ... }
if (p != nullptr and p->elem == x) { ... }
```

👍 ?:, and i or sí són mandroses.

```
λ> f x y = x
λ> a = 2
λ> b = 0
λ> f a (div a b)
👉 2
```

👍 (div a b) no és avaluat.

```
λ> f x y = x
λ> h x = h x
λ> f 3 (h 0)
👉 3
```

👍 h mai és avaluada.

Llistes infinites

Generació de la llista infinita de zeros

```
zeros :: [Int]
-- amb repeat
zeros = repeat 0

-- amb cycle
zeros = cycle [0]

-- amb iterate
zeros = iterate id 0

-- amb recursivitat infinita
zeros = 0 : zeros

-- prova
λ> take 6 zeros
👉 [0, 0, 0, 0, 0, 0]
```

Llistes infinites

Generació de la llista infinita de naturals

```
naturals :: [Int]

-- amb rangs infinits
naturals = [0..]

-- amb iterate
naturals = iterate (+1) 0

-- amb recursivitat infinita
naturals = 0 : map (+1) naturals

-- prova
λ> take 6 naturals
👉 [0, 1, 2, 3, 4, 5]
```

Llistes infinites

Generació de la llista infinita de factorials

```
factorials :: [Integer]
factorials = scanl (*) 1 [1..]

λ> take 6 $ scanl (*) 1 [1..]
👉 [1, 1, 2, 6, 24, 120]
```

Llistes infinites

Generació de la llista infinita de nombres de Fibonacci

Llistes infinites

Generació de la llista infinita de nombres de Fibonacci

```
fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

Llistes infinites

Generació de la llista infinita de nombres de Fibonacci

```
fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

```
fibs :: [Integer]
fibs = fibs' 0 1
  where
    fibs' m n = m : fibs' n (m+n)
```


Llistes infinites

Generació dels nombres primers amb el Garbell d'Eratòstenes

Llistes infinites

Generació dels nombres primers amb el Garbell d'Eratòstenes

```
primers :: [Integer]
primers = garbell [2..]
  where
    garbell (p : xs) = p : garbell [x | x <- xs, x `mod` p /= 0]
```

Llistes infinites

Generació dels nombres primers amb el Garbell d'Eratòstenes

```
primers :: [Integer]

primers = garbell [2..]
  where
    garbell (p : xs) = p : garbell [x | x <- xs, x `mod` p /= 0]
```

Perspectiva: C++

```
// retorna la llista de tots els nombres primers ≤ n.
vector<int> primers (int n) {
    vector<int> ps;
    vector<bool> p(n+1, true);
    for (int i = 2; i <= n; ++i) {
        if (p[i]) {
            ps.push_back(i);
            for (int j = 2*i; j <= n; j += i) {
                p[j] = false;
            }
        }
    }
    return ps;
}
```

Avaluació ansiosa

En Haskell es pot forçar cert nivell d'avaluació ansiosa (*eager*) usant l'operador infix `$!`.

`f $! x` avalua primer `x` i després `f x` però només avalua fins que troba un constructor.

Sumari

- L'**avaluació mandrosa** (*lazy*) només avalua el que cal.
- El compilador avalua mandrosament usant *thunks*: valors pendents d'avaluar.
- L'avaluació mandrosa permet definir i manipular objectes infinits.
- Programar amb objectes infinits és més fàcil que amb objectes finits: no cal controlar els finals!

Exercicis Lab 3

Feu aquests problemes de Jutge.org:

- [P93588](#) Usage of comprehension lists
- [P98957](#) Infinite lists

Programació Funcional en Haskell



Tipus

Jordi Petit

Departament de Ciències de la Computació
Universitat Politècnica de Catalunya

Tipus predefinitos

Ja hem vist que existeixen una sèrie de tipus predefinitos:

- Tipus simples:

- Int, Integer, Float, Double
- Bool
- Char

- Tipus estructurats:

- Llistes
- Tuples
- Funcions

```
5    :: Integer
True :: Bool
'a'  :: Char
[1,2,3] :: [Integer]
('b',4) :: (Char,Integer)
not    :: Bool -> Bool
```

Tots els identificadors de tipus comencem amb majúscula.

Tipus polimòrfics

```
length :: [a] -> Int
map    :: (a -> b) -> [a] -> [b]
```

El **polimorfisme paramètric** és un mecanisme senzill que permet definir funcions (i tipus) que s'escriuen genèricament, sense dependre dels tipus dels objectes sobre els quals s'apliquen.

En Haskell, les **variables de tipus** poden prendre qualsevol valor i estan quantificades universalment. Per convenció `a`, `b`, `c`, ...

Tipus polimòrfics

Per a utilitzar funcions amb tipus polimòrfics cal que hi hagi una substitució de les variables de tipus que s'adeqüi a l'aplicació que estem fent.

Exemple: `map even [3,6,1]` té tipus `[Bool]` ja que:

- el tipus de `map` és `(a -> b) -> [a] -> [b]`,
- el tipus de `even` és `Int -> Bool`,
- per tant, `a` es pot substituir per `Int` i `b` es pot substituir per `Bool`,
- i el tipus final de l'expressió és `[Bool]`.

Una expressió dóna error de tipus si no existeix una substitució per a les seves variables de tipus.

Exemple: `map not ['b','c']` dóna error de tipus ja que:

- per una banda, `a` hauria de ser `Bool`,
- per altre banda, `a` hauria de ser `Char`.

Tipus sinònims

La construcció `type` permet substituir un tipus (complex) per un nou nom.

Els dos tipus són intercanviables.

```
type Euros = Float  
sou :: Persona -> Euros
```

```
type Diccionari = String -> Int  
crear :: Diccionari  
cercar :: Diccionari -> String -> Int  
inserir :: Diccionari -> String -> Int -> Diccionari  
esborrar :: Diccionari -> String -> Diccionari
```

Els tipus sinònims aporten claredat (però no més seguretat).

💡 Per a més seguretat, mireu `newtype` (no el considerem).

Tipus enumerats

Els **tipus enumerats** donen la llista de valors possibles dels objectes d'aquell tipus.

```
data Jugada = Pedra | Paper | Tisores

data Operador
  = Suma
  | Resta
  | Producte
  | Divisio

data Bool = False | True    -- predefinit
```

Els valors enumerats (**constructors**), han de començar amb majúscula.

Els tipus enumerats es poden desconstruir amb patrons:

```
guanya :: Jugada -> Jugada -> Bool
-- diu si la primera jugada guanya a la segona

guanya Paper Pedra = True
guanya Pedra Tisores = True
guanya Tisores Paper = True
guanya _ _ = False
```

Tipus algebraics

Els **tipus algebraics** defineixen diversos constructors, cadascun amb zero o més dades associades.

```
data Forma
= Rectangle Float Float    -- alçada, amplada
| Quadrat Float           -- mida
| Cercle Float            -- radi
| Punt
```

Les dades es creen especificant el constructor i els seus valors respectius:

```
λ> r = Rectangle 3 4
λ> :type r
☞ r :: Forma

λ> c = Cercle 2.0
λ> :type c
☞ c :: Forma
```

Tipus algebraics

```
data Forma
= Rectangle Float Float    -- alçada, amplada
| Quadrat Float           -- mida
| Cercle Float             -- radi
| Punt
```

Els tipus algebraics es poden desconstruir amb patrons:

```
area :: Forma -> Float

area (Rectangle amplada alçada) = amplada * alçada
area (Quadrat mida) = area (Rectangle mida mida)
area (Cercle radi) = pi * radi^2
area Punt = 0
```

```
λ> area (Rectangle 3 4)
👉 12
```

```
λ> c = Cercle 2.0
λ> area c
👉 12.566370614359172
```

Tipus algebraics

Per escriure valors algebraics, cal afegir `deriving (Show)` al final del tipus.
⇒ més endavant veurem què vol dir.

```
data Punt = Punt Int Int
    deriving (Show)

data Rectangle = Rectangle Punt Punt
    deriving (Show)
```

```
λ> p1 = Punt 2 3
λ> p1
👉 Punt 2 3

λ> p2 = Punt 4 6
λ> p2
👉 Punt 4 6

λ> r = Rectangle p1 p2
λ> r
👉 Rectangle (Punt 2 3) (Punt 4 6)
```

Sumari

- Haskell disposa de tipus predefinitos bàsics per enters, reals, booleans, caràcters.
- El polimorfisme paramètric permet definir funcions (i tipus) genèrics usant variables de tipus.
- Els tipus enumerats donen la llista de valors possibles dels objectes d'aquell tipus.
- Els tipus algebraics generalitzen els tipus enumerats, proveïnt diferents constructors amb dades associades.

Programació Funcional en Haskell



Aplicacions de tipus algebraics

Jordi Petit

Departament de Ciències de la Computació
Universitat Politècnica de Catalunya

Arbres binaris d'enters

Els tipus algebraics també es poden definir recursivament!

```
data Arbin = Buit | Node Int Arbin Arbin
  deriving (Show)
```

```
λ> a1 = Node 1 Buit Buit
λ> a2 = Node 2 Buit Buit
λ> a3 = Node 3 a1 a2
λ> a4 = Node 4 a3 Buit
λ> a4
👉 Node 4 (Node 3 (Node 1 Buit Buit) (Node 2 Buit Buit)) Buit

λ> a5 = Node 5 a4 a4      -- I 💜 sharing
λ> a5
👉 Node 5 (Node 4 (Node 3 (Node 1 Buit Buit) (Node 2 Buit Buit)) Buit) (Node 4 (Node 3 (Node 1 Buit Buit) (Node 2 Buit Buit)) Buit) Buit
```

Com sempre, la desconnexió via patrons marca el camí: 🦶

```
alcada :: Arbin -> Int

alcada Buit = 0
alcada (Node _ fe fd) = 1 + max (alcada fe) (alcada fd)
```

Arbres binaris genèrics

Els tipus algebraics també tenen polimorfisme paramètric!

```
data Arbin a = Buit | Node a (Arbin a) (Arbin a)
  deriving (Show)
```

```
a1 :: Arbin Int
a1 = Node 3 (Node 1 Buit Buit) (Node 2 Buit Buit)

a2 :: Arbin Forma
a2 = Node (Rectangle 3 4) (Node (Cercle 2) Buit Buit) (Node Punt Buit Buit)
```

```
alcada :: Arbin a -> Int

alcada Buit = 0
alcada (Node _ fe fd) = 1 + max (alcada fe) (alcada fd)
```

```
preordre :: Arbin a -> [a]

preordre Buit = []
preordre (Node x fe fd) = [x] ++ preordre fe ++ preordre fd
```

Arbres generals genèrics

```
data Argal a = Argal a [Argal a]      -- (no hi ha arbre buit en els arbres generals)
  deriving (Show)
```

```
a = Argal 4 [Argal 1 [], Argal 2 [], Argal 3 [Argal 0 []]]
```

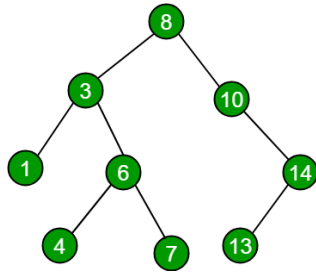
```
mida :: Argal a -> Int
```

```
mida (Argal _ fills) = 1 + sum (map mida fills)
```

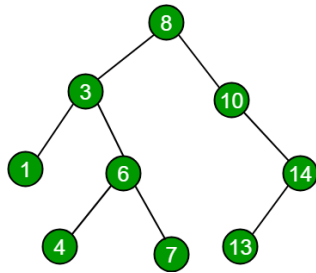
```
preordre :: Argal a -> [a]
```

```
preordre (Argal x fills) = x : concatMap preordre fills
```

Arbres binaris de cerca

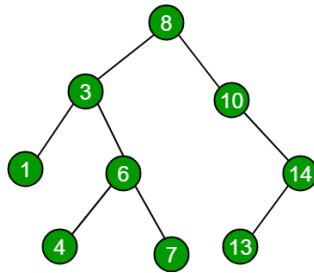


Arbres binaris de cerca



```
Abc a = Buit | Node a (Abc a) (Abc a) -- arbre binari de cerca
```

Arbres binaris de cerca



```
Abc a = Buit | Node a (Abc a) (Abc a)      -- arbre binari de cerca
```

```
buit      :: Abc a      -- retorna un arbre buit
cerca     :: Ord a => a -> Abc a -> Bool    -- diu si un arbre conté un element
insereix  :: Ord a => a -> Abc a -> Abc a    -- inserció d'un element
esborra   :: Ord a => a -> Abc a -> Abc a    -- esborrat d'un element (exercici)
```

Arbres binaris de cerca

```
Abc a = Buit | Node a (Abc a) (Abc a)    -- arbre binari de cerca

buit :: Abc a                             -- retorna un arbre buit
buit = Buit

cerca :: Ord a => a -> Abc a -> Bool       -- diu si un arbre conté un element

cerca x Buit = False
cerca x (Node k fe fd)
  | x < k      = cerca x fe
  | x > k      = cerca x fd
  | x == k     = True

insereix :: Ord a => a -> Abc a -> Abc a   -- inserció d'un element

insereix x Buit = Node x Buit Buit
insereix x (Node k fe fd)
  | x < k      = Node k (insereix x fe) fd
  | x > k      = Node k fe (insereix x fd)
  | x == k     = Node k fe fd

esborra :: Ord a => a -> Abc a -> Abc a    -- esborrat d'un element (exercici)
```


Expressions booleans amb variables

```
data ExprBool
  = Val Bool
  | Var Char
  | Not ExprBool
  | And ExprBool ExprBool
  | Or ExprBool ExprBool
  deriving (Show)

type Dict = Char -> Bool
```

```
eval :: ExprBool -> Dict -> Bool
```

```
eval (Val x) d = x
eval (Var v) d = d v
eval (Not e) d = not $ eval e d
eval (And e1 e2) d = eval e1 d && eval e2 d
eval (Or e1 e2) d = eval e1 d || eval e2 d
```

```
e = (And (Or (Val False) (Var 'x')) (Not (And (Var 'y') (Var 'z'))))
d = (`elem` "xz")
eval e d
-- evalua (F ∨ x) ∧ (¬ (y ∧ z)) amb x = z = T i y = F
```

Perspectiva

```
data Expr a
= Val a
| Var String
| Neg (Expr a)
| Sum (Expr a) (Expr a)
| Res (Expr a) (Expr a)
| Mul (Expr a) (Expr a)
| Div (Expr a) (Expr a)
```

Com seria en C++?

Perspectiva

```
template <typename a> class Expr {  
    struct ValData {  
        a x;  
    };  
  
    struct VarData {  
        string v;  
    };  
  
    struct NegData {  
        Node* e;  
    };  
  
    struct OpData {  
        Node* e1;  
        Node* e2;  
    };  
  
    enum Constructor {Val, Var, Neg,  
                     Sum, Res, Mul, Div};
```

```
struct Node {  
    Constructor c;  
    union {  
        ValData val;  
        VarData var;  
        NegData neg;  
        OpData op;  
    };  
  
    Node* p; // punter al node amb l'expressi  
  
public:  
  
    Expr ExprVal (const a& x);  
    Expr ExprVar (const string& v);  
    Expr ExprNeg (const Expr& e);  
    Expr ExprSum (const Expr& e1,  
                  const Expr& e2);  
  
    ...  
};
```

Perspectiva

```
template <typename a> class Expr {  
    struct ValData {  
        a x;  
    };  
  
    struct VarData {  
        string v;  
    };  
  
    struct NegData {  
        Node* e;  
    };  
  
    struct OpData {  
        Node* e1;  
        Node* e2;  
    };  
  
    enum Constructor {Val, Var, Neg,  
                     Sum, Res, Mul, Div};
```

```
struct Node {  
    Constructor c;  
    union {  
        ValData val;  
        VarData var;  
        NegData neg;  
        OpData op;  
    };  
  
    Node* p; // punter al node amb l'expressi  
  
public:  
    Expr ExprVal (const a& x);  
    Expr ExprVar (const string& v);  
    Expr ExprNeg (const Expr& e);  
    Expr ExprSum (const Expr& e1,  
                  const Expr& e2);  
    ...  
};
```

I encara falten les operacions i la gestió de la memòria!



Programació Funcional en Haskell



Tipus generèrics predefinit

Jordi Petit

Departament de Ciències de la Computació
Universitat Politècnica de Catalunya

Llistes genèriques

```
data Llista a = Buida | a `DavantDe` (Llista a)
```

```
l1 = 3 `DavantDe` 2 `DavantDe` 4 `DavantDe` Buida
```

```
llargada :: Llista a -> Int
```

```
llargada Buida = 0
```

```
llargada (cap `DavantDe` cua) = 1 + llargada cua
```

Llistes genèriques

```
data Llista a = Buida | a `DavantDe` (Llista a)
```

```
l1 = 3 `DavantDe` 2 `DavantDe` 4 `DavantDe` Buida
```

```
llargada :: Llista a -> Int
```

```
llargada Buida = 0
```

```
llargada (cap `DavantDe` cua) = 1 + llargada cua
```

Les llistes de Haskell són exactament això! (amb una mica de sucre sintàctic 🍬)

```
data [a] = [] | a : [a]
```

```
l1 = 3:2:4:[]    -- l1 = [3, 2, 4]
```

```
length :: [a] -> Int
```

```
length [] = 0
```

```
length (x:xs) = 1 + length xs
```

Maybe a

El tipus polimòrfic `Maybe a` està predefinit així:

```
data Maybe a = Just a | Nothing
```

Expressa dues possibilitats:

- la presència d'un valor (de tipus `a` amb el constructor `Just`), o
- la seva absència (amb el constructor buit `Nothing`).

Aplicacions:

- Indicar possibles valor nuls.
- Indicar absència d'un resultat.
- Reportar un error.

Exemples: (busqueu doc a [Hoogle](#))

```
find :: (a -> Bool) -> [a] -> Maybe a
-- cerca en una llista amb un predicat

lookup :: Eq a => a -> [(a,b)] -> Maybe b
-- cerca en una llista associativa
```


Either a b

El tipus polimòrfic `Either a b` està predefinit així:

```
data Either a b = Left a | Right b
```

Expressa dues possibilitats per un valor:

- un valor de tipus `a` (amb el constructor `Left`), o
- un valor de tipus `b` (amb el constructor `Right`).

Aplicacions:

- Indicar que un valor pot ser, alternativament, de dos tipus.
- Reportar un error. Habitualment:
 - `a` és un `String` i és el diagnòstic de l'error.
 - `b` és del tipus del resultat esperat.
 - **Mnemotècnic:** *right* vol dir *dreta* i també *correcte*.

Exemple:

```
secDiv :: Float -> Float -> Either String Float
secDiv _ 0 = Left "divisió per zero"
secDiv x y = Right (x / y)
```

Programació Funcional en Haskell



Classes de tipus

Jordi Petit

Departament de Ciències de la Computació
Universitat Politècnica de Catalunya

Classes de tipus

Una **classe de tipus** (*type class*) és una interfície que defineix un comportament.

Els tipus poden **instanciar** (implementar seguint la interfície) una o més classes de tipus.

La instanciació es pot fer

- automàticament pel compilador per a certes classes predefinides, o
- a mà.

Les classes de tipus

- són la forma de tenir sobrecàrrega en Haskell, i
- propocionen una altra forma de polimorfisme.

⚠ Les classes de tipus de Haskell no són classes de OOP com a C++ o Java (més aviat són com els `interfaces` de Java).

La classe Eq

La funció `elem` necessita comparar elements per igualtat:

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

La declaració `(Eq a) =>` indica que els tipus `a` sobre els quals es pot aplicar la funció `elem` han de ser instàncies de la classe `Eq`.

La classe predefinida `Eq` dóna operacions d'igualtat i desigualtat:

```
class Eq a where  
    (==) :: a -> a -> Bool  
    (/=) :: a -> a -> Bool
```

I fins i tot ja proporciona definicions per defecte (circulars, què hi farem!):

```
class Eq a where  
    (==) :: a -> a -> Bool  
    (/=) :: a -> a -> Bool  
  
x == y = not (x /= y)  
x /= y = not (x == y)
```

La classe Eq

El nostre tipus `Jugada` (encara) no dóna suport a la classe `Eq`:

```
data Jugada = Pedra | Paper | Tisora

λ> Paper /= Paper
error: "No instance for (Eq Jugada) arising from a use of ‘/=’"

λ> Pedra `elem` [Paper, Pedra, Paper]
error: "No instance for (Eq Jugada) arising from a use of ‘elem’"
```

Amb `deriving (Eq)` demanem al compilador que instanciï automàticament la classe `Eq` (usant igualtat estructural):

```
data Jugada = Pedra | Paper | Tisora
  deriving (Eq)

λ> Paper /= Paper
👉 False

λ> Pedra `elem` [Paper, Pedra, Paper]
👉 True
```

La classe Eq

Per alguns tipus, la igualtat estructural no és suficient:

```
data Racional = Racional Int Int -- numerador, denominador
  deriving (Eq)

λ> Racional 3 2 == Racional 6 4
👎 False
```

En aquests casos cal instanciar la classe a mà:

```
instance Eq Racional where
  (Racional n1 d1) == (Racional n2 d2) = n1 * d2 == n2 * d1

λ> Racional 3 2 == Racional 6 4
👍 True

λ> Racional 3 2 /= Racional 6 4
👍 False
```

Només cal definir `==` perquè la definició per defecte de `/=` ja ens convé.

La classe Eq

Per alguns tipus, instanciar una classe també requereix alguna altra classe:

```
data Arbin a = Buit | Node a (Arbin a) (Arbin a)

instance Eq a => Eq (Arbin a) where

    Buit == Buit = True
    (Node x1 fe1 fd1) == (Node x2 fe2 fd2) = x1 == x2 && fe1 == fe2 && fd1 == fd2
    _ == _ = False
```

Informació sobre instàncies

Amb la comanda `:info T` (o `:i T`) de l'interpret es pot veure de quines classes és instància un tipus `T`:

```
λ> :i Racional
data Racional = Racional Int Int
instance Eq Racional

λ> :i Int
data Int = GHC.Types.I# GHC.Prim.Int#
instance Eq Int
instance Ord Int
instance Show Int
instance Read Int
instance Enum Int
instance Num Int
instance Real Int
instance Bounded Int
instance Integral Int
```


La classe Ord

La classe predefinida `Ord` (que requereix la classe `Eq`) dóna operacions d'ordre:

```
data Ordering = LT | EQ | GT           -- possibles resultats d'una comparació d'ordre

class (Eq a) => Ord a where
  compare :: a -> a -> Ordering
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min :: a -> a -> a

  compare x y
    | x == y    = EQ
    | x <= y    = LT
    | otherwise = GT
  x <  y = compare x y == LT
  x >  y = compare x y == GT
  x <= y = compare x y /= GT
  x >= y = compare x y /= LT
```

El mínim que cal per fer la instanciació és definir el `<=` o el `compare`.

Tot i que no es verifica, s'espera que les instàncies d'`Ord` compleixin aquestes lleis:

- Transitivitat: si `x <= y` && `y <= z` llavors `x <= z`.
- Reflexivitat: `x <= x`.
- Antisimetria: si `x <= y` && `y <= x` llavors `x == y`.

La classe Show

La classe predefinida `Show` dóna suport per convertir valors en textos:

```
class Show a where  
  show :: a -> String
```

Amb `deriving (Show)`, el compilador la ofereix automàticament (usant sintàxi Haskell):

```
data Racional = Racional Int Int      -- numerador, denominador  
  deriving (Eq, Show)  
  
λ> show $ Racional 3 2  ➡ "Racional 3 2"  
λ> show $ Racional 6 4  ➡ "Racional 6 4"  💔
```

Alternativament, per fer la instanciació a mà només cal definir el `show`:

```
instance Show Racional where  
  show (Racional n d) = (show $ div n m) ++ "/" ++ (show $ div d m)  
  where m = gcd n d  
  
λ> show $ Racional 3 2  ➡ "3/2"  
λ> show $ Racional 6 4  ➡ "3/2"  💖
```

La classe Read

La classe predefinida `Read` dóna suport per convertir textos en valors:

```
class Read a where  
  read :: String -> a
```

Amb `deriving (Read)`, el compilador la ofereix automàticament (usant sintàxi Haskell).

Alternativament, per fer la instanciació a mà només cal definir el `read`.

Compte: Al usar `read`, sovint cal especificar el tipus de retorn, perquè el compilador sàpiga a quin de tots els `reads` sobrecarregats ens referim:

```
λ> read "38"  
λ> (read "38") :: Int      ⚡ "Exception: Prelude.read: no parse"  
λ> (read "38") :: Integer  📎 38  
λ> (read "38") :: Float   📎 38  
λ> (read "38") :: Float   📎 38.0
```

La classe Num

La classe predefinida `Num` dona suport a operadors aritmètics bàsics:

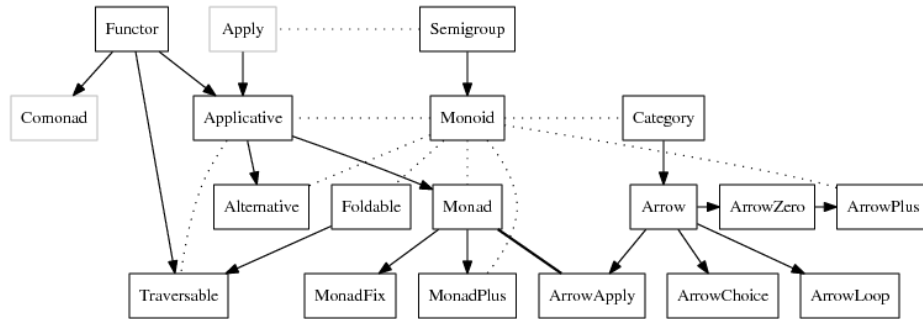
```
class (Eq a, Show a) => Num a where
  (+), (-), (*)      :: a -> a -> a
  negate, abs, signum :: a -> a
  fromInteger        :: Integer -> a

  x - y      = x + negate y
  negate x = 0 - x
```

Per fer la instanciació cal definir totes les operacions menys `negate` o `-`.

Els tipus `Int`, `Integer`, `Float` i `Double` són instàncies de la classe `Num`.

Altres classes predefinides



Imatge: <https://wiki.haskell.org/Typeclassopedia>

Ús de classes en declaracions de tipus

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

Quin és el tipus de `suma`?

Ús de classes en declaracions de tipus

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

Quin és el tipus de `suma`?

```
suma :: [Int] -> Int
```

Ús de classes en declaracions de tipus

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

Quin és el tipus de `suma`?

```
suma :: [Int] -> Int
```

✗ més general!

Ús de classes en declaracions de tipus

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

Quin és el tipus de `suma`?

```
suma :: [Int] -> Int
```

✗ més general!

```
suma :: [a] -> a
```

Ús de classes en declaracions de tipus

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

Quin és el tipus de `suma`?

```
suma :: [Int] -> Int
```

✗ més general!

```
suma :: [a] -> a
```

✗ el tipus `a` no pot ser qualsevol: ha de tenir l'operació `+`!

Ús de classes en declaracions de tipus

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

Quin és el tipus de `suma`?

```
suma :: [Int] -> Int
```

✗ més general!

```
suma :: [a] -> a
```

✗ el tipus `a` no pot ser qualsevol: ha de tenir l'operació `+`!

```
suma :: Num a => [a] -> a
```

Ús de classes en declaracions de tipus

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

Quin és el tipus de `suma`?

```
suma :: [Int] -> Int
```

✗ més general!

```
suma :: [a] -> a
```

✗ el tipus `a` no pot ser qualsevol: ha de tenir l'operació `+`!

```
suma :: Num a => [a] -> a
```

✓ el tipus `a` ha de ser instància de `Num`!

Ús de classes en declaracions de tipus

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

Quin és el tipus de `suma`?

```
suma :: [Int] -> Int
```

✗ més general!

```
suma :: [a] -> a
```

✗ el tipus `a` no pot ser qualsevol: ha de tenir l'operació `+`!

```
suma :: Num a => [a] -> a
```

✓ el tipus `a` ha de ser instància de `Num`!

Les condicions sobre les variables de tipus es posen davant de `=>` a la signatura.

El sistema de tipus de Haskell és capaç d'inferir tipus i condicions automàticament.
 \Rightarrow més endavant veurem com.

Definició de classes pròpies

Només cal utilitzar la mateixa sintàxi que ja hem vist.

Exemple: Classe per a predicats.

```
class Pred a where
  sat  :: a -> Bool
  unsat :: a -> Bool

  unsat = not . sat
```

Instanciació pels enters:

```
instance Pred Int where
  sat 0 = False
  sat _ = True
```

Instanciació pels arbres binaris:

```
instance Pred a => Pred (Arbin a) where
  sat Built = True
  sat (Node x fe fd) = sat x && sat fe && sat fd
```

Exercicis Sessió 4

Feu aquests problemes de Jutge.org:

- [P97301](#) FizzBuzz
- [P37072](#) Arbre binari
- [P80618](#) Cua (1)

Programació Funcional en Haskell



Functors

Jordi Petit

Departament de Ciències de la Computació
Universitat Politècnica de Catalunya

Functors

Ja sabem aplicar funcions:

```
λ> (+3) 2      ➡ 5
```

Però...

```
λ> (+3) (Just 2)
```



En aquest cas, podem fer servir `fmap`!

```
λ> fmap (+3) (Just 2)      ➡ Just 5
λ> fmap (+3) Nothing      ➡ Nothing
```

I també funciona amb `Either`, llistes, tuples i funcions:

```
λ> fmap (+3) (Right 2)      ➡ Right 5
λ> fmap (+3) (Left "err")   ➡ Left "err"

λ> fmap (+3) [1, 2, 3]      ➡ [4, 5, 6]      -- igual que map
λ> fmap (+3) (1, 2)         ➡ (1, 5)          -- perquè (,) és un tipus
λ> (fmap (*2) (+1)) 3        ➡ 8               -- igual que (.)
```

Functors

`fmap` aplica una funció als elements d'un contenidor genèric `f a` retornant un contenidor del mateix tipus.

`fmap` és una funció de les instàncies de la classe `Functor`:

```
λ> :type fmap
fmap :: Functor f => (a -> b) -> (f a -> f b)
```

On

```
λ> :info Functor
class Functor f where
  fmap :: (a -> b) -> (f a -> f b)
```

Maybe és functor

El tipus `Maybe` és instància de `Functor`:

```
λ> :info Maybe
data Maybe a = Nothing | Just a
instance Ord a => Ord (Maybe a)
instance Eq a => Eq (Maybe a)
instance Applicative Maybe
instance Functor Maybe
instance Monad Maybe
⋮
```

Concretament,

```
instance Functor Maybe where
  fmap f Nothing = Nothing
  fmap f (Just x) = Just (f x)
```

Aplicació

Consulta a una BD:

- Llenguatge sense Maybe:

```
post = Posts.find(1234)
if post is None:
    return None
else:
    return post.title
```

- En Haskell:

```
fmap getPostTitle (findPost 1234)
```

o també:

```
getPostTitle `fmap` findPost 1234
```

o millor (<\$> és l'operador infix per a fmap): (es llegeix *fmap*)

```
getPostTitle <$> findPost 1234
```

Either a és functor

El tipus `Either a` és instància de `Functor`:

```
instance Functor (Either a) where
  fmap f (Left  x) = Left  x
  fmap f (Right x) = Right (f x)
```

Fixeu-vos que `Either` té dos paràmetres:

- el tipus `Either a` és la instància de `Functor`,
- el tipus `Either` no.

Les llistes són functors

El tipus [] (llista) és instància de Functor:

```
instance Functor [] where
  fmap = map           -- potser és al revés, poc importa
```

Les fonctions són functors

Les funcions també són instàncies de `Functor`:

```
instance Functor ((->) r) where  
  fmap = (.)
```

Exemple:

```
λ> (*3) <$> (+2) <$> Just 1      ↵ Just 9  
λ> (*3) <$> (+2) <$> Nothing    ↵ Nothing
```

Lleis dels functors

Les instàncies de functors han de tenir aquestes propietats:

1. Identitat: `fmap id ≡ id`.
2. Composició: `fmap (g1 . g2) ≡ fmap g1 . fmap g2`.

Nota: Haskell no verifica aquestes propietats (però les pot utilitzar), és responsabilitat del programador fer-ho.

Exercici: Comproveu que `Maybe`, `Either a`, `[]`, `(,)` i `(->)` compleixen les lleis dels functors.

Arbres binaris com a functors

Instànciació pròpia dels functors pels arbres binaris:

```
data Arbin a
  = Buit
  | Node a (Arbin a) (Arbin a)
  deriving (Show)
```

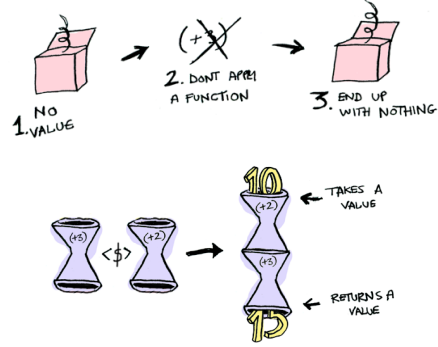
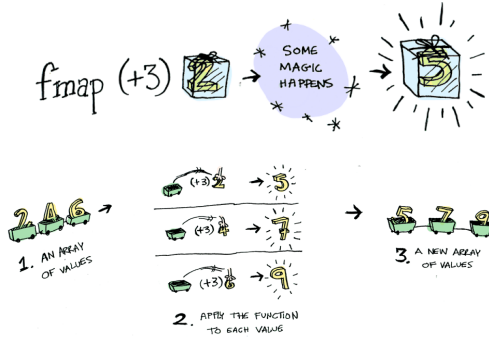
```
instance Functor (Arbin) where
  fmap f Buit = Buit
  fmap f (Node x fe fd) = Node (f x) (fmap f fe) (fmap f fd)
```

```
a = Node 3 Buit (Node 2 (Node 1 Buit Buit) (Node 1 Buit Buit))
λ> fmap (*2) a
👉 Node 6 Buit (Node 4 (Node 2 Buit Buit) (Node 2 Buit Buit))
λ> fmap even a
👉 Node False Buit (Node True (Node False Buit Buit) (Node False Buit Buit))
```

Exercici: Comproveu que `Arbin` compleix les lleis dels functors.

Sumari

La classe `Functor` captura la idea de tipus contenidor genèric al qual es pot aplicar una funció als seus elements per canviar el seu contingut (però no el contenidor).



Programació Funcional en Haskell



Aplicatius

Jordi Petit

Departament de Ciències de la Computació
Universitat Politècnica de Catalunya

Aplicatius

Ja sabem aplicar funcions:

I ho sabem fer sobre contenidors:

```
λ> (+3) 2
```



5

```
λ> fmap (+3) (Just 2)
```



Just 5

Però què passa si la funció és en un contenidor?

```
λ> (Just (+3)) (Just 2)
```



En aquest cas, podem fer servir `<*>!` (es llegeix *app*)

```
λ> Just (+3) <*> Just 2
λ> Just (+3) <*> Nothing
λ> Nothing <*> Just (+3)
λ> Nothing <*> Nothing

λ> Right (+3) <*> Right 2
λ> Right (+3) <*> Left "err"
λ> Left "err" <*> Right 2
λ> Left "err1" <*> Left "err2"

λ> [(*2), (+2)] <*> [1, 2, 3]
```



Just 5



Nothing



Nothing



Nothing



Right 5



Left "err"



Left "err"



Left "err1 "



[2, 4, 6, 3, 4, 5]

Aplicatius

L'operador `<*>` és una operació de la classe `Applicative` (que també ha de ser functor):

```
class Functor f => Applicative f where
  (<*>) :: f (a -> b) -> (f a -> f b)
  pure  :: a -> f a
```

- `<*>` aplica una funció dins d'un contenidor a uns valors dins d'un contenidor. Els contenidors són genèrics i del mateix tipus.
- `pure` construeix un contenidor amb un valor.

Lleis dels aplicatius

Les instàncies d'aplicatius han de tenir aquestes propietats:

1. Identitat:

```
pure id <*> v ≡ v.
```

2. Homomorfisme:

```
pure f <*> pure x ≡ pure (f x).
```

3. Intercanvi:

```
u <*> pure y ≡ pure ($ y) <*> u.
```

4. Composició:

```
u <*> (v <*> w) ≡ pure (.) <*> u <*> v <*> w.
```

5. Relació amb el functor:

```
fmap g x ≡ pure g <*> x.
```

Instanciacions d'aplicatius

Maybe és aplicatiu:

```
instance Applicative Maybe where
  pure = Just
  Nothing <*> _ = Nothing
  Just f <*> x = fmap f x
```

Either a és aplicatiu:

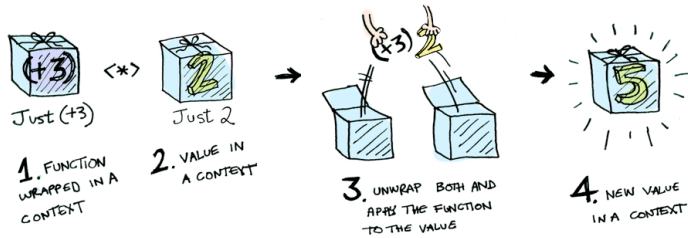
```
instance Applicative (Either a) where
  pure = Right
  Left x <*> _ = Left x
  Right f <*> x = fmap f x
```

Exercici: Instancieu les llistes com a aplicatius. Hi ha dues formes de fer-ho.

Sumari

Els aplicatius permeten aplicar funcions dins d'un contenidor a objectes dins del mateix contenidor.

- `pure` construeix un contenidor amb un valor.
- `<*>` aplica una funció dins d'un contenidor a uns valors dins d'un contenidor:



Programació Funcional en Haskell



Mònades

Jordi Petit

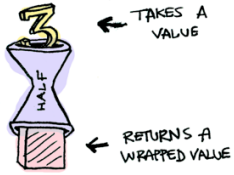
Departament de Ciències de la Computació
Universitat Politècnica de Catalunya

Mònades

Considerem que `meitat` és una funció que només té sentit sobre parells:

```
meitat :: Int -> Maybe Int  
  
meitat x  
| even x    = Just (div x 2)  
| otherwise = Nothing
```

Podem veure la funció així: Donat un valor, retorna un valor empaquetat.



Però llavors no li podem ficar valors empaquetats!



Mònades

Cal una funció que desempaqueti, apliqui `meitat` i deixi empaquetat.

Aquesta funció es diu `>>=` (es llegeix *bind*)

```
λ> Just 40 >>= meitat      ➡ Just 20
λ> Just 31 >>= meitat      ➡ Nothing
λ> Nothing >>= meitat      ➡ Nothing

λ> Just 20 >>= meitat >>= meitat      ➡ Just 5
λ> Just 20 >>= meitat >>= meitat >>= meitat ➡ Nothing
```

L'operador `>>=` és una operació de la classe `Monad`:

```
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  -- i més coses
```

El tipus `Maybe` és instància de `Monad`:

```
instance Monad Maybe where
  Nothing >>= f = Nothing
  Just x  >>= f = f x
```

Mònades

De fet, les mònades tenen tres operacions:

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b

  r >> k    =  r >>= (\_ -> k)
```

- `return` empaqueta.
- `>>=` desempaqueta, aplica i empaqueta.
- `>>` és purament estètica.

Instanciacions

Els tipus `Maybe`, `Either a` i `[]` són instàncies de `Monad`:

```
instance Monad Maybe where
    return      = Just
    Nothing >=> f = Nothing
    Just x  >=> f = f x

instance Monad (Either a) where
    return      = Right
    Left x  >=> f = Left x
    Right x >=> f = f x

instance Monad [] where
    return x      = [x]
    xs >=> f       = concatMap f xs
```

Lleis de les mònades

Les instàncies de mònades han de tenir aquestes propietats:

1. Identitat per l'esquerra:

```
return x >>= f ≡ f x.
```

2. Identitat per la dreta:

```
m >>= return ≡ m.
```

3. Associativitat:

```
(m >>= f) >>= g ≡ m >>= (\x -> f x >>= g).
```

Nota: Haskell no verifica aquestes propietats (però les pot utilitzar), és responsabilitat del programador fer-ho.

Exercici: Comproveu que `Maybe`, `Either a i []` compleixen les lleis de les mònades.

Notació `do`

La **notació `do`** és sucre sintàctic per facilitar l'ús de les mònades.
⇒ Amb `do`, codi funcional *sembla* codi imperatiu amb assignacions.

Els còmputs es poden **seqüenciar**:

```
do { e1 ; e2 }
```

≡

```
do  
  e1  
  e2
```

≡

```
e1 >> e2
```

≡

```
e1 >>= \_ -> e2
```

I amb `<-` **extreure** el seus resultats:

```
do { x <- e1 ; e2 }
```

≡

```
do  
  x <- e1  
  e2
```

≡

```
e1 >>= \x -> e2
```

Notació do: Exemple

Tenim llistes associatives amb informació sobre propietaris de cotxes, les seves matrícules, els seus models i les seves etiquetes d'emissions:

```
data Model = Seat127 | TeslaS3 | NissanLeaf | ToyotaHybrid deriving (Eq, Show)

data Etiqueta = Eco | B | C | Cap deriving (Eq, Show)

matricules = [("Joan", 6524), ("Pere", 6332), ("Anna", 5313), ("Laia", 9999)]

models = [(6524, NissanLeaf), (6332, Seat127), (5313, TeslaS3), (7572, ToyotaHybrid)]

etiquetes = [(Seat127, Cap), (TeslaS3, Eco), (NissanLeaf, Eco), (ToyotaHybrid, B)]
```

Donat un nom de propietari, volem saber quina és la seva etiqueta d'emissions:

```
etiqueta :: String -> Maybe Etiqueta
```

És `Maybe` perquè, potser el propietari no existeix, o no tenim la seva matrícula, o no tenim el seu model, o no tenim la seva etiqueta...

Ens anirà bé usar aquesta funció predefinida de cerca:

```
lookup :: Eq a => a -> [(a, b)] -> Maybe b
```


Notació do: Exemple

Solució amb `case`: 🐻

```
etiqueta nom =  
  case lookup nom matricules of  
    Nothing -> Nothing  
    Just mat -> case lookup mat models of  
                  Nothing -> Nothing  
                  Just mod -> lookup mod etiquetes
```

Notació do: Exemple

Solució amb `case`: 🤖

```
etiqueta nom =  
  case lookup nom matricules of  
    Nothing -> Nothing  
    Just mat -> case lookup mat models of  
                  Nothing -> Nothing  
                  Just mod -> lookup mod etiquetes
```

Solució amb notació `do`: 💜

```
etiqueta nom = do  
  mat <- lookup nom matricules  
  mod <- lookup mat models  
  lookup mod etiquetes
```

Notació **do**: Exemple

Amb notació **do**:

```
etiqueta nom = do  
  mat <- lookup nom matricules  
  mod <- lookup mat models  
  lookup mod etiquetes
```

Notació **do**: Exemple

Amb notació **do**:

```
etiqueta nom = do  
  mat <- lookup nom matricules  
  mod <- lookup mat models  
  lookup mod etiquetes
```

Transformació de notació **do** a funcional:

```
etiqueta nom =  
  lookup nom matricules >=> \mat -> lookup mat models >=> \mod -> lookup mod etiquetes
```

Notació **do**: Exemple

Amb notació **do**:

```
etiqueta nom = do
  mat <- lookup nom matricules
  mod <- lookup mat models
  lookup mod etiquetes
```

Transformació de notació **do** a funcional:

```
etiqueta nom =
  lookup nom matricules >>= \mat -> lookup mat models >>= \mod -> lookup mod etiquetes
```

Amb un format diferent queda clara l'equivalència: 🤪

```
etiqueta nom =
  lookup nom matricules >>= \mat ->
  lookup mat models >>= \mod ->
  lookup mod etiquetes
```

Funcions predefinides per a mònades

Moltes funcions predefinides tenen una extnsió per la classe `Monad`:

- `mapM`, `filterM`, `foldM`, `zipWithM`, ...

També disposem d'operacions per estendre (*lift*) operacions per treballar amb elements de la classe `Monad`. S'han d'importar:

```
import Control.Monad
liftM  :: Monad m => (a -> b) -> m a -> m b
liftM2 :: Monad m => (a -> b -> c) -> m a -> m b -> m c
```

Per exemple, podem crear una funció per suma `Maybes`:

```
sumaMaybes :: Num a => Maybe a -> Maybe a -> Maybe a
sumaMaybes = liftM2 (+)

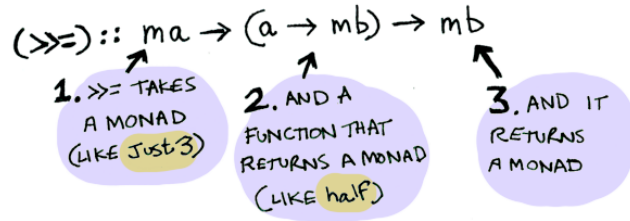
λ> sumaMaybes (Just 3) (Just 4)  ➡ Just 7
```

O fer-ho directament:

```
λ> liftM2 (+) (Just 3) (Just 4)  ➡ Just 7
```

Sumari (1)

- Les mònades permeten aplicar una funció que retorna un valor en un contenidor a un valor en un contenidor.



- Molts tipus predefinitos són instàncies de mònades.
- La notació `do` simplifica l'ús de les mònades.

Sumari (2)

- Aplicacions:
 - IO
 - Parsers
 - Logging
 - Estat mutable
 - No determinisme
 - Paral·lelisme
- Lectura recomanada: [Monads for functional programming](#) de P. Wadler.

Programació Funcional en Haskell



Entrada/Sortida

Jordi Petit

Departament de Ciències de la Computació
Universitat Politècnica de Catalunya

Entrada/Sortida

L'entrada/sortida en Haskell es basa en una mònada:

- El programa principal és `main :: IO ()`
- S'usa el constructor de tipus `IO` per gestionar l'entrada/sortida.
- `IO` és instància de `Monad`.
- Es sol usar amb notació `do`.

Algunes operacions bàsiques:

```
getChar    :: IO Char      -- obté següent caràcter
getLine    :: IO String    -- obté següent línia
getContents :: IO String    -- obté tota l'entrada

putChar     :: Char -> IO () -- escriu un caràcter
putStr      :: String -> IO () -- escriu un text
putStrLn    :: String -> IO () -- escriu un text i un salt de línia
print       :: Show a => a -> IO () -- escriu qualsevol showable
```

`()` és una tupla de zero camps i `()` és l'únic valor de tipus `()`.
(\Leftrightarrow void de C).

Hello world!

```
main = do
  putStrLn "Com et dius?"
  nom <- getLine
  putStrLn $ "Hola " ++ nom + "!"
```

Compilació i execució:

```
> ghc programa.hs
[1 of 1] Compiling Main          ( programa.hs, programa.o )
Linking programa ...

> ./programa
Com et dius?
Jordi
Hola Jordi!
```

Del revés

```
main = do
  x <- getLine
  let y = reverse x
  putStrLn x
  putStrLn y
```

Compilació i execució:

```
> ghc programa.hs
[1 of 1] Compiling Main                ( programa.hs, programa.o )
Linking programa ...

> ./programa
GAT
GAT
TAG
```

Exemple

Llegir seqüència de línies acabades en * i escriure cadascuna del revés:

```
main = do
  line <- getline
  if line /= "*" then do
    putStrLn $ reverse line
    main
  else
    return ()
```

Exemple

Llegir seqüència de línies i escriure cadascuna del revés:

```
main = do
  contents <- getContents
  mapM (putStrLn . reverse) (lines contents)
```

L'E/S també és *lazy*, no cal preocupar-se perquè l'entrada sigui massa llarga.

Solució alternativa:

```
main = interact reverse
```

- `interact` és una acció predefinida d'ordre superior que va llegint línies i per a cadascuna escriu el resultat de cridar el paràmetre sobre ella.

```
interact :: (String -> String) -> IO ()
```

let i where en notació do

Degut a la definició del `>>=`, el `where` pot donar problemes:

```
main = do
  x <- getLine
  print f
  where f = factorial (read x)

✗ error: Variable not in scope: x :: String
```

Si ho escrivim amb `>>=`, tenim

```
main = getLine >>= \x -> print f
  where f = factorial (read x)
```

que no pot ser, ja que a les definicions del `where` no podem usar la variable abstracta `x`.


Amb el `do` cal usar el `let` (sense `in`):

```
main = do
  x <- getLine
  let f = factorial (read x)
  print f
```


Alternativament (més lleig):

```
main = do
  x <- getLine
  f <- return $ factorial (read x)
  print f
```

Intuició sobre la mónada IO

Podem veure l'entrada/sortida com funcions que modifiquen el món: `món1`  `món2`.

Les operacions d'entrada/sortida reben un món i retornen un món.

Cadascuna s'encadena amb l'anterior, com un relleu. 

Exemple: Llegir i escriure dos caràcters.

```
data World = ... -- descripció del món

myGetChar :: World -> (World, Char)

myPutChar :: Char -> World -> (World, ())

myMain :: World -> (World, ())

myMain w0 = let (w1, c1) = myGetChar w0
                (w2, c2) = myGetChar w1
                (w3, ()) = myPutChar c1 w2
                (w4, ()) = myPutChar c2 w3
            in (w4, ())
```

(1) Passant el relleu.

Intuició sobre la mónada IO

Podem veure l'entrada/sortida com funcions que modifiquen el món: món1 \rightsquigarrow món2.

Les operacions d'entrada/sortida reben un món i retornen un món.

Cadascuna s'encadena amb l'anterior, com un relleu. 🏊

Exemple: Llegir i escriure dos caràcters.

```
data World = ... -- descripció del món

myGetChar :: World -> (World, Char)

myPutChar :: Char -> World -> (World, ())

myMain :: World -> (World, ())

myMain w0 = let (w1, c1) = myGetChar w0
                (w2, c2) = myGetChar w1
                (w3, ()) = myPutChar c1 w2
                (w4, ()) = myPutChar c2 w3
            in (w4, ())
```

(1) Passant el relleu.

```
type IO a = World -> (World, a)

getChar :: IO Char

putChar :: Char -> IO ()

main :: IO ()

main =
    getChar >=> \c1 ->
    getChar >=> \c2 ->
    putChar c1 >>
    putChar c2
```

(2) Fent que IO sigui instància de Monad.

Intuició sobre la mónada IO

Podem veure l'entrada/sortida com funcions que modifiquen el món: món1 \rightsquigarrow món2.

Les operacions d'entrada/sortida reben un món i retornen un món.

Cadascuna s'encadena amb l'anterior, com un relleu. 🏊

Exemple: Llegir i escriure dos caràcters.

```
type IO a = World -> (World -> a)

getChar :: IO Char

putChar :: Char -> IO ()

main :: IO ()

main =
  getChar >>= \c1 ->
  getChar >>= \c2 ->
  putChar c1 >>
  putChar c2
```

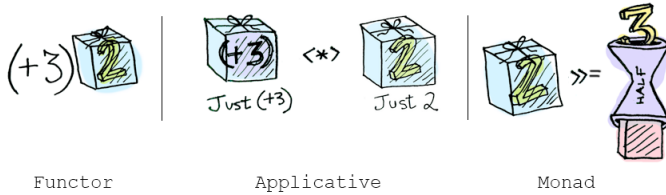
```
main = do
  c1 <- getChar
  c2 <- getChar
  putChar c1
  putChar c2
```

(2) Fent que IO sigui instància de Monad.

(3) Usant notació do.

Sumari

- Hem vist tres classes predefinides molt importants en Haskell: Functors, Aplicatius, Mònades.



- Molts tipus predefinits són instàncies d'aquestes classes: `Maybe`, `Either`, llistes, tuples, funcions, `IO`, ...
- La notació `do` simplifica l'ús de les mònades.
- La classe `IO` permet disposar d'entrada/sortida en un llenguatge funcional pur.

Final

L'**estat d'un programa** descriu tota la informació que no és local a una funció en particular. Això inclou:

- variables globals
- entrada
- sortida

Pensar sobre un programa amb estat és difícil perquè:

- L'estat perviu d'una crida d'una funció a una altra.
- L'estat és a l'abast de totes les funcions.
- L'estat és mutable.
- L'estat canvia en el temps.
- Cap funció és responsable de l'estat.

Estat: 🤖

Sense estat: 💜

Les mònades no eliminen la noció d'estat en un programa, però eliminen la necessitat de mencionar-lo.

Exercicis sessió 5

Feu aquests problemes de Jutge.org:

- Functors, aplicatius i mònades:
 - [P70540](#) Expressions
 - [P50086](#) Cua 2
 - [P58738](#) Arbres amb talla
- Entrada/sortida:
 - [P87974](#) Hola / Adéu
 - [P87082](#) Índex massa corporal

