

Llenguatges de Programació

## Sessió 1: Recursivitat



Jordi Petit

UNIVERSITAT POLITÈCNICA DE CATALUNYA  
BARCELONATECH

Facultat d'Informàtica de Barcelona



# Contingut

- [Aperitiu](#)
- Eines
- Tipus bàsics
- Funcions
- Tuples
- Llistes
- Funcions habituals en llistes
- Exercicis

# 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
```

```
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
```

```
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)
```

```
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]
```

```
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 = 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
```



# Contingut

- Aperitiu
- Eines
- Tipus bàsics
- Funcions
- Tuples
- Llistes
- Funcions habituals en llistes
- Exercicis

# Eines necessàries

Glasgow Haskell Compiler (GHC):

- compilador (ghc)
- intèrpret (ghci)

Editor de codi

Terminal

Jutge

# Instal·lació del GHC

**Linux i Mac:**

Utilitzeu el `ghcup` ([referència](#)):

```
curl --proto '=https' --tlsv1.2 -sSf https://get-ghcup.haskell.org | sh
```

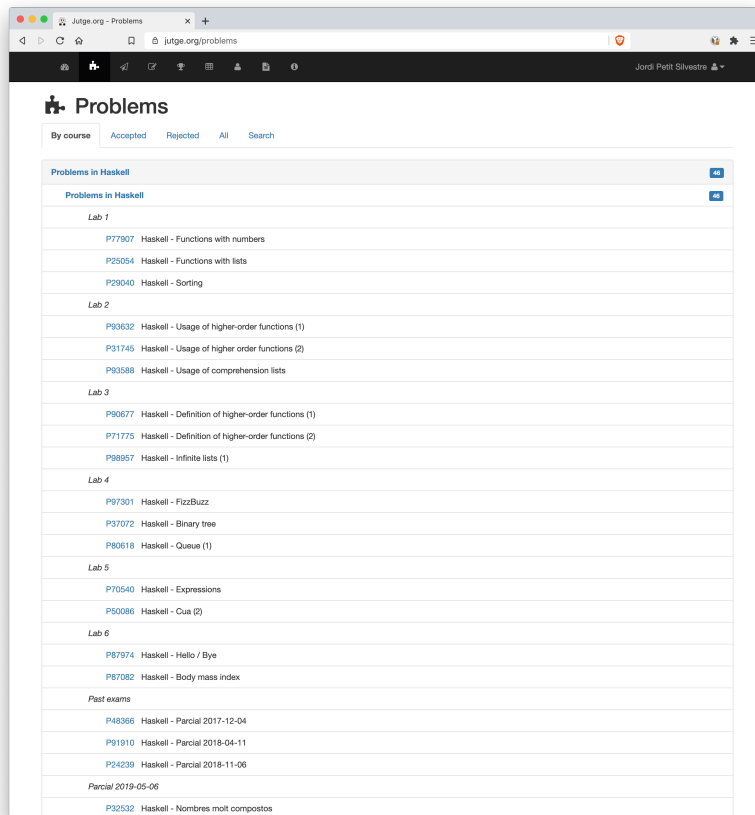
**Windows:**

Seguiu aquest [video](#).

Font: [GHCup: How to install](#)



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



# Comandes de l'interpret

Intèrpret:

ghci.

Comandes més usuals:

Comanda	Exemple	Descripció
:load	:l arxiu	càrrega un script
:quit	:q	sortida de l'interpret
:reload	:r	recarrega l'últim arxiu carregat
:type	:t 3	tipus de l'expressió
:info	:i []	informació associada al paràmetre (útil a partir del tema de classes)
:sprint		visualització dels <i>thunks</i> (útil per l'avaluació mandrosa)
:help		ajuda

# Contingut

- Aperitiu
- Eines
- Tipus bàsics
- Funcions
- Tuples
- Llistes
- Funcions habituals en llistes
- Exercicis

# Booleans

Tipus: Bool

Literals: False i True

Operacions:

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

Exemples:

<code>not True</code>	👉 False
<code>not False</code>	👉 True
<code>True    False</code>	👉 True
<code>True &amp;&amp; False</code>	👉 False
<code>(False    True) and True</code>	👉 True
<code>not (not True)</code>	👉 True
<code>not not True</code>	❌ -- 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:

<code>3 + 4 * 5</code>	 <code>23</code>
<code>(3 + 4) * 5</code>	 <code>35</code>
<code>(3 + 4) * 5</code>	 <code>35</code>
<code>2^10</code>	 <code>1024</code>
<code>3 + 1 /= 4</code>	 <code>False</code>
<code>div 11 2</code>	 <code>5</code>
<code>mod 11 2</code>	 <code>1</code>
<code>rem 11 2</code>	 <code>1</code>
<code>mod (-11) 2</code>	 <code>1</code>



# 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

Font: Haskell report

# 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 múltiple:

```
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
```

# Contingut

- Aperitiu
- Eines
- Tipus bàsics
- Funcions
- Tuples
- Llistes
- Funcions habituals en llistes
- Exercicis

# 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 dóna la seva declaració de tipus (capçalera).
2. Després es dóna 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 -> 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).

## 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'interpret:

```
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

<b>div</b> 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`.

# Contingut

- Aperitiu
- Eines
- Tipus bàsics
- Funcions
- Tuples
- Llistes
- Funcions habituals en llistes
- Exercicis

# Tuples

Una tupla és un tipus estructurat que permet desar diferents valors de tipus  $t_1, t_2, \dots, t_n$  en un únic valor de tipus  $(t_1, t_2, \dots, t_n)$ .

- 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")      📌 3  
snd (3, "rave")      📌 "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.

# Contingut

- Aperitiu
- Eines
- Tipus bàsics
- Funcions
- Tuples
- Llistes
- Funcions habituals en llistes
- Exercicis

# 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

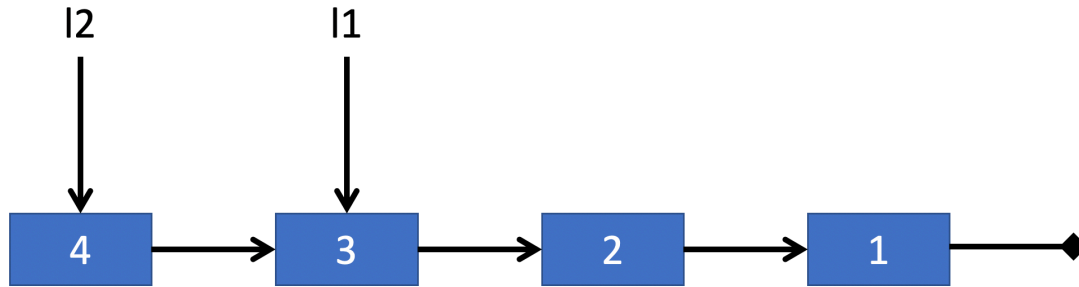
```
16 : (12 : (21 : []))
```

# 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  $2 : (5 : 8 : [])$  substituint  $x$  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*  $[2, 5, 8]$  substituint  $x1$  amb 2,  $x2$  amb 5 i  $xs$  amb  $[8]$ .
- $x1:x2:xs$  *matches*  $[2, 5]$  substituint  $x1$  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
```

# Contingut

- Aperitiu
- Eines
- Tipus bàsics
- Funcions
- Tuples
- Llistes
- Funcions habituals en llistes
- Exercicis

# 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:

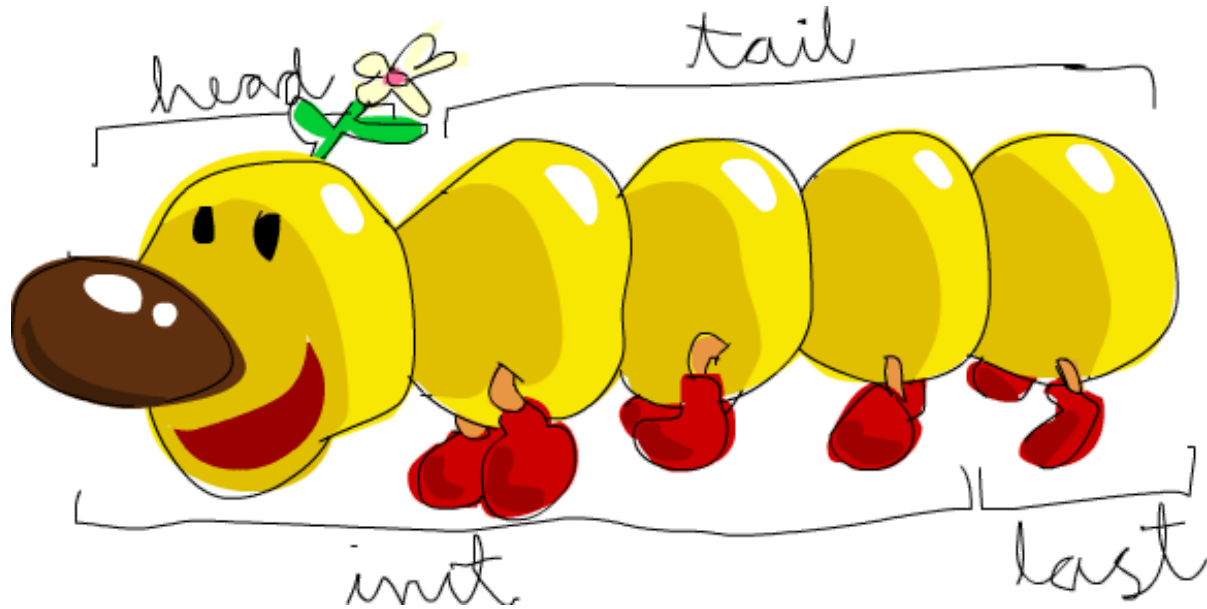
- `tail xs` és la llista `xs` sense el seu primer element.
- `init xs` és la llista `xs` sense el seu darrer element.

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]
```

# Contingut

- Aperitiu
- Eines
- Tipus bàsics
- Funcions
- Tuples
- Llistes
- Funcions habituals en llistes
- Exercicis

# Exercicis

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. Veredicte 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.

