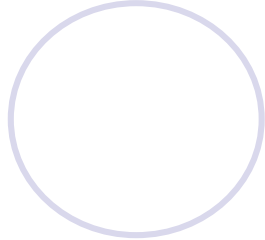
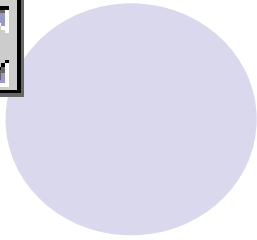


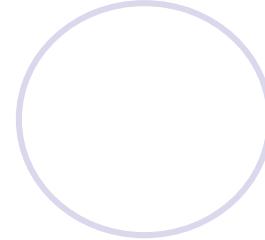
Tema 7

Estructuras infinitas y
cíclicas





Evaluación perezosa



evaluación de expresiones

orden normal
(*call-by-name*)

orden aplicativo
(*call-by-value*)

+

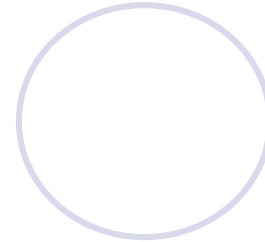
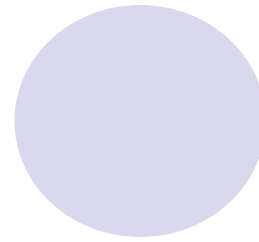
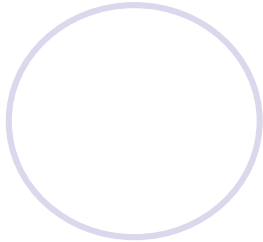
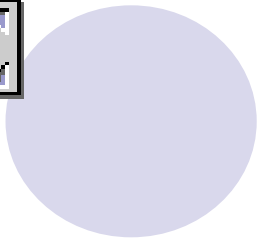
||

no repetición
de evaluaciones

evaluación ansiosa

||

evaluación perezosa
(*call-by-need*)



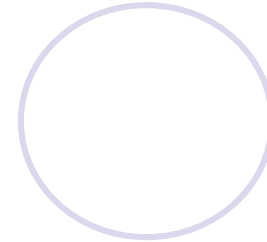
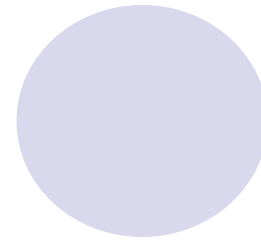
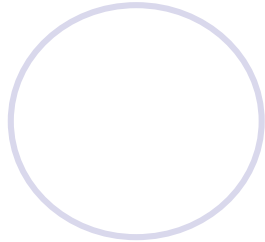
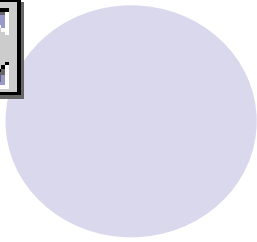
- Orden normal: se reduce la sub-expresión reducible:

- más externa (*outermost*)

$$\frac{g \ (f1 \ e1) \ (f2 \ e2)}{(f1 \ e1 \ (f2 \ e2 \ e3))}$$

- más a la izquierda (*leftmost*)

$$\frac{f1 \ e1}{+} \quad f2 \ e2$$



$$g(x, y) = g(y, x)$$

$$h(x) = g(x, x)$$

$$r(x) = 5$$

- Orden normal: *Se evalúan sólo los argumentos que la definición de la función requiere para obtener el resultado.*

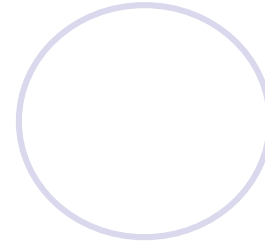
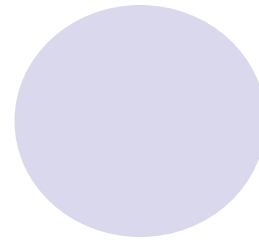
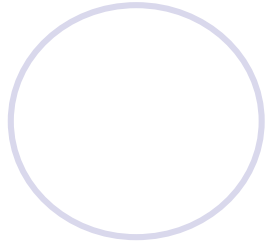
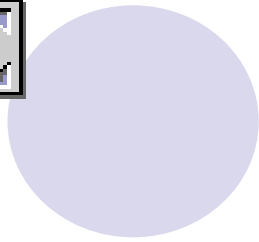
$$r(h(1)) \Rightarrow 5$$

- Orden aplicativo: *Se evalúan siempre primero los argumentos antes de aplicar la función.*

$$r(h(1)) \Rightarrow r(g(1, 1))$$

$$\Rightarrow r(g(1, 1))$$

$$\Rightarrow \dots$$



- *El orden normal obtiene la forma normal, siempre que esta existe (Teorema II de Church-Rosser).*

$$\begin{aligned} g(1, 1) &\Rightarrow g(1, 1) \\ &\Rightarrow \dots \end{aligned}$$

- *El orden aplicativo puede no encontrar la forma normal de la expresión, aunque exista.*



- Repetición de cálculos: *inconveniente que se debe evitar para usar el orden normal de modo eficiente.*

$$g(x, y) = x * y$$

$$h(x) = g(x, x)$$

$$h(5+3) \Rightarrow g(5+3, 5+3)$$

$$\Rightarrow (\underline{5+3}) * (\underline{5+3})$$

$$\Rightarrow 8 * (\underline{5+3})$$

$$\Rightarrow 8 * 8$$

$$\Rightarrow 64$$

Evaluación perezosa = Orden normal + no repetición



Patrones, pereza y listas infinitas



- El ajuste de patrones también se hace de forma perezosa: Para decidir si una expresión ajusta con un patrón se evalúa hasta extraer los constructores necesarios y no más.
- Ejemplo:

```
head (map (*2) [1..])  
⇒ head (map (*2) (1:[2..])  
⇒ head ((*2)1 : map cuadrado [2..])  
⇒ (*2)1  
⇒ 2 1
```



Más ejemplos de ajuste de patrones perezoso



```
filter (<10) [8..]
```

```
⇒ 8: filter (<10) [9..]
```

```
⇒ 8: 9: filter (<10) [10..]
```

```
⇒ 8: 9: filter (<10) [11..]
```

```
⇒ 8: 9: filter (<10) [12..]
```

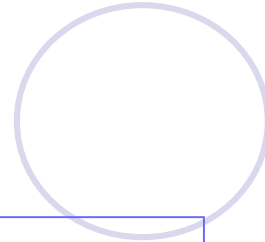
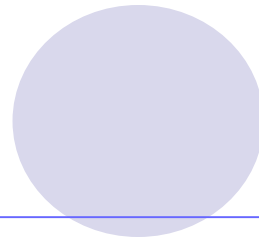
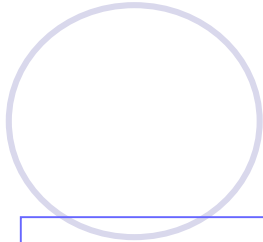
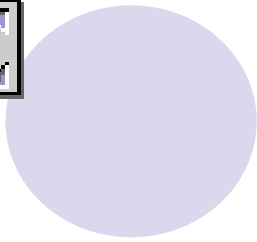
```
⇒ [8,9, ...
```

```
takeWhile (<10) [8..]
```

```
⇒ 8: takeWhile (<10) [9..]
```

```
⇒ 8: 9: takeWhile (<10) [10..]
```

```
⇒ 8: 9: []
```

```
pot n = [n^i | i <- [0..]]
```

```
take 2 (pot 2)
```

```
⇒ take 2 (2^0 : [2^i | i <- [1..]])
```

```
⇒ 2^0 : take 1 [2^i | i <- [1..]]
```

```
→ Output: [1
```

```
⇒ 1 : take 1 (2^1 : [2^i | i <- [2..]])
```

```
⇒ 1 : 2^1 : take 0 [2^i | i <- [2..]]
```

```
→ Output: [1,2
```

```
⇒ 1 : 2 : []
```

```
→ Output: [1,2]
```



La función predefinida `iterate`



- Definición informal:

`iterate f x = [x, f x, f2 x, f3 x, ...]`

`iterate (*2) 2 \Rightarrow [2, 4, 8, 16, ...]`

- Puede usarse para definir la notación de secuencias aritméticas

`[m..] = iterate (+1) m`

`[m..n] = takeWhile (<=n) (iterate (+1) m)`

- Definición formal

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

`iterate f x = x : iterate f (f x)`



Ejemplo 1: Dígitos de un número



digitos 39425 = [3,9,4,2,5]

39425

↓ iterate (`div` 10)

[39425,3942,394,39,3,0,0,0,...]

↓ takeWhile (/=0)

[39425,3942,394,39,3]

↓ map (`mod` 10)

[5,2,4,9,3]

↓ reverse

[3,9,4,2,5]

```
digitos = reverse . map (`mod` 10) .  
          takeWhile (/=0) . iterate (`div` 10)
```



Ejemplo 2: La criba de Eratóstenes



	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	...



La función `criba` en Haskell



`criba [2..]`

\Rightarrow `2 : criba [3, 5, 7, 9, 11, 13, 15, 17, 19, ...]`

\Rightarrow `2 : 3 : criba [5, 7, 11, 13, 17, 19, ...]`

\Rightarrow `2 : 3 : 5 : criba [7, 11, 13, 17, 19, ...]`

`...`

```
criba :: Integral a => [a] -> [a]
criba (x:xs) = x : criba (filter ((/= 0).(`mod`x)) xs)

primos :: [Integer]
primos = criba [2..]
```



primos

⇒ criba [2..]

2:[3..]

3:[4..]

⇒ 2 : (criba (filter (nomult 2) [3..]))

⇒ 2 : (criba (3 : (filter (nomult 2) [4..])))

⇒ 2:3:(criba (filter (nomult 3) (filter (nomult 2) [4..])))

⇒ 2:3:(criba (filter (nomult 3) (filter (nomult 2) [5..])))

⇒ 2:3:(criba (filter (nomult 3) (5:filter (nomult 2) [6..]))))

⇒ 2:3:(criba (5:(filter (nomult 3)(filter (nomult 2) [6..])))))

⇒ 2:3:5:(criba (filter (nomult 5)

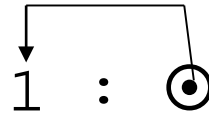
(filter (nomult 3)(filter (nomult 2) [6..]))))



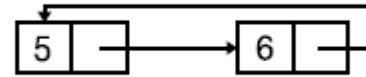
Estructuras cíclicas



• `unos = 1 : unos`



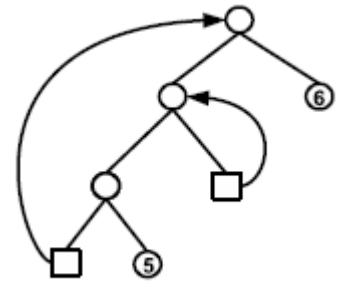
• `clist = 5 : 6 : clist`



• `data Arbol a = Hoja a | Nodo (Arbol a) (Arbol a)`

`deriving Show`

`ca = let x = (Nodo (Nodo ca (Hoja 5)) x)`
`in Nodo x (Hoja 6)`



```
unfold :: (Show a1, Eq a2, Num a2) => a2 -> Arbol a1 -> Arbol String
unfold _ (Hoja r) = Hoja (show r)
unfold 0 (Nodo ai ad) = Hoja " "
unfold n (Nodo ai ad) = Nodo (unfold (n-1) ai) (unfold (n-1) ad)
```



La función predefinida `repeat`



- `repeat x = [x,x,x,...`
- `unos` es equivalente a `repeat 1`
- Definición correcta: `repeat x = x : repeat x`
- Pero para que se cree una estructura cíclica:
`repeat x = xs where xs = x:xs`

Ejercicio: Evaluar `repeat 1` de las dos formas



iterate definida de forma cíclica



- ```
iterate f x = zs
 where zs = x : map f zs
```

- ```
iterate (2*) 1
```

\Rightarrow `zs` where `zs = 1 : map (2*) zs`

\Rightarrow `1 : map (2*)` \odot

\Rightarrow `1 :` \downarrow `2 : map (2*)` \odot

\Rightarrow `1 :` `2 :` \downarrow `4 : map (2*)` \odot

\Rightarrow ...



iterate sin where no es cíclica



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

```
iterate (2*) 1
```

```
⇒ 1 : map (2*) (iterate (2*) 1)
```

```
⇒ 1 : map (2*) (1: map (2*) (iterate (2*) 1))
```

```
⇒ 1: 2 : map (2*) (map (2*)
```

```
    (1: map (2*) (iterate (2*) 1)))
```

```
⇒ 1: 2 : 4 : map (2*) (map (2*)
```

```
    ( map (2*) (iterate (2*) 1)))
```

```
⇒ ...
```



Ejemplo: Los números de Hamming



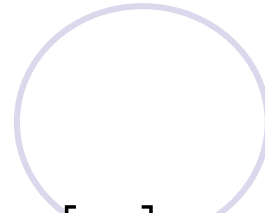
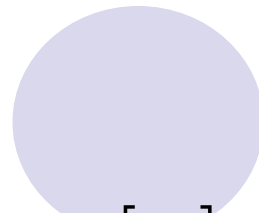
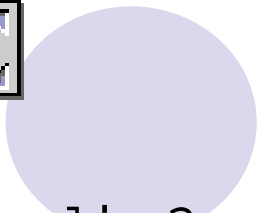
- 1 es un número de Hamming
- Si x es un número de Hamming, entonces $2x$, $3x$ y $5x$ también son

$1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, \dots$

```
hamming :: [Integer]
```

```
-- hamming es la lista infinita creciente de  
-- los números de Hamming
```

```
hamming = 1: fundir3 (map (2*) hamming)  
                  (map (3*) hamming)  
                  (map (5*) hamming)
```



```
fundir3 :: Ord a => [a] -> [a] -> [a] -> [a]
-- Pre: xs ys y zs son listas ordenadas crecientemente

-- (fundir3 xs ys zs) es la lista ordenada creciente
-- sin repeticiones que resulta de mezclar xs, ys y zs

fundir3 xs ys zs =
  fundir2 xs (fundir2 ys zs)
  where
    fundir2 (x:xs) (y:ys)
      | x == y = x: fundir2 xs ys
      | x < y  = x: fundir2 xs (y:ys)
      | x > y  = y: fundir2 (x:xs) ys
```



Ejemplos de uso



```
Main> takeWhile (<100) hamming
```

```
[1,2,3,4,5,6,8,9,10,12,15,16,18,20,24,25,27,30,32,36,  
40,45,48,50,54,60,64,72,75,80,81,90,96] :: [Integer]
```

```
Main> take 33 hamming
```

```
[1,2,3,4,5,6,8,9,10,12,15,16,18,20,24,25,27,30,32,36,  
40,45,48,50,54,60,64,72,75,80,81,90,96] :: [Integer]
```

```
Main> dropWhile (<100) hamming
```

```
[100,108,120,125,128,135,144,150,160,162,180,192,200,  
216,225,240,243,250,256,270,288,300,320,324,360,375,  
384,400,405,432,450,480,486,500,512,540,576,600,625,  
640,648,675,720,729,750,768,800,810,864,900,960,972,  
1000,1024,1080,1125,1152,1200,1215,1250,1280,  
{Interrupted!}]
```



Estructuras cíclicas en ejecución



hamming

\Rightarrow 1: fundir3 (map (2*) \odot)
(map (3*) \odot) (map (5*) \odot)

\Rightarrow 1: fundir3 (2*1 : map (2*) \odot)
(3*1 : map (3*) \odot)
(5*1 : map (5*) \odot)



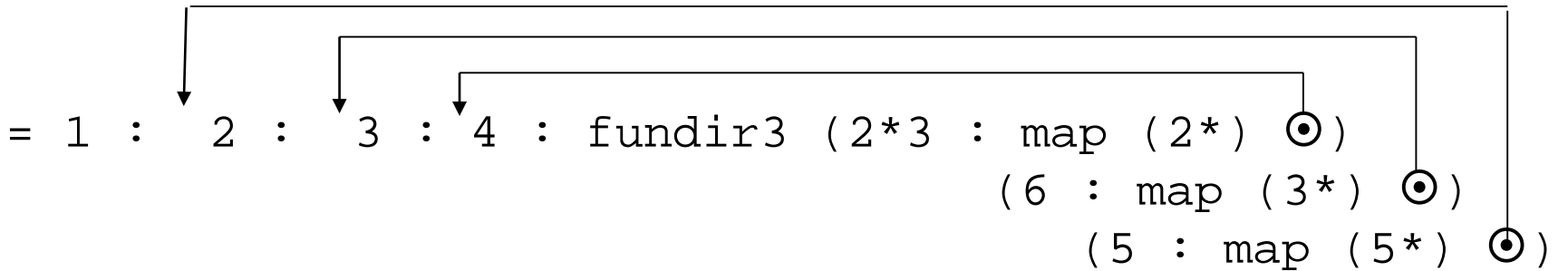
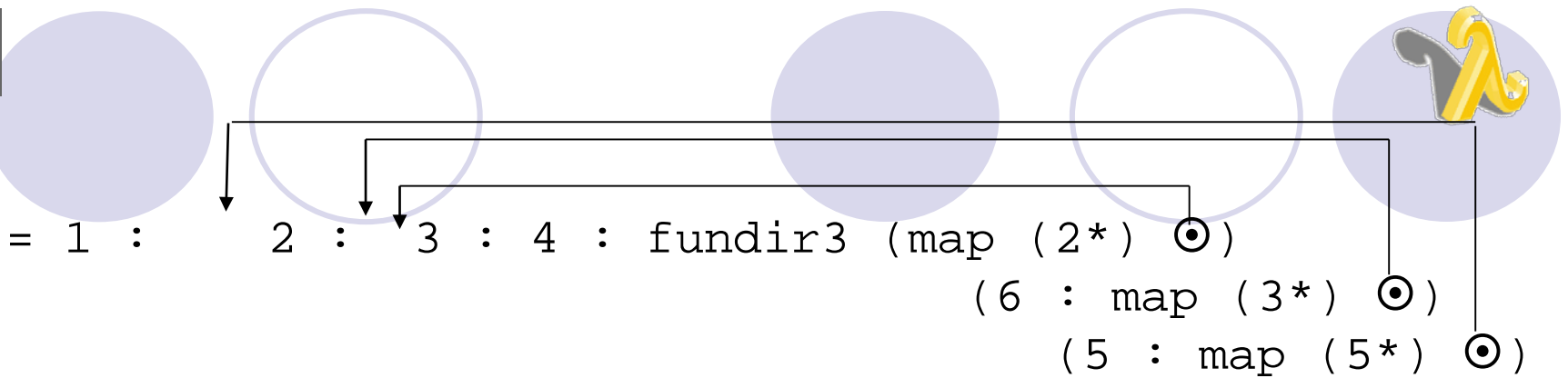
= 1 :
2 : fundir3 (map (2*) ⊙)
 (3 : map (3*) ⊙)
 (5 : map (5*) ⊙)



= 1 :
2 : fundir3 (2*2 : map (2*) ⊙)
 (3 : map (3*) ⊙)
 (5 : map (5*) ⊙)

= 1 :
2 : 3 : fundir3 (4 : map (2*) ⊙)
 (map (3*) ⊙)
 (5 : map (5*) ⊙)

= 1 :
2 : 3 : fundir3 (4 : map (2*) ⊙)
 (3*2 : map (3*) ⊙)
 (5 : map (5*) ⊙)



Ejercicio: Continuar la evaluación hasta obtener el 8