Tema 11. Listas infinitas

11.1 Listas infinitas

✓ La evaluación perezosa permite trabajar con listas infinitas

✓ Algunas funciones predefinidas construyen listas infinitas:

```
-- repeat v ⟹ [ v, v, v, ... ]
repeat :: a → [a]
repeat x = xs where xs = x : xs
```

```
-- cycle [ v1, v2, ..., vn ] ⟹ [ v1, v2, ..., vn, v1, v2, ..., vn, ... ]
cycle :: [a] → [a]
cycle [] = error "Prelude.cycle : empty list"
cycle xs = xs' where xs' = xs ++ xs'
```

```
-- iterate f x ⟹ [ x, f x, f (f x), f (f (f x)), ... ]
iterate :: (a → a) → a → [a]
iterate f x = x : iterate f (f x)
```

✓ Se pueden definir listas infinitas utilizando estas funciones:

```
losNaturales :: Num a ⇒ [a]
losNaturales = iterate (+1) 0

múltiplosDe :: Num a ⇒ a → [a]
múltiplosDe x = iterate (+x) 0

potenciasDe :: Num a ⇒ a → [a]
potenciasDe x = iterate (*x) 1
```

Argumentos infinitos y resultados finitos

✓ La evaluación perezosa permite obtener resultados finitos aunque los argumentos sean infinitos:

? $take\ 10\ (múltiplosDe\ 3)$
$[0, 3, 6, 9, 12, 15, 18, 21, 24, 27] \ :: \ [Integer]$

La evaluación es:

$$
\begin{array}{lll}
take & :: & Int \to [a] \to [a] \\
take\ 0 \quad \_ & = & [\,] \\
take\ \_ \quad [\,] & = & [\,] \\
take\ (n+1)\ (x : xs) & = & x \ : \ take\ n\ xs
\end{array}
$$

$$iterate\ f\ x \ = \ x \ : \ iterate\ f\ (f\ x)$$

$$múltiplosDe\ x \ = \ iterate\ (+x)\ 0$$

$take\ 2\ (múltiplosDe\ 3)$
$\Longrightarrow$ {por $múltiplosDe$ ya que $take$ no puede evaluarse}
$take\ 2\ (iterate\ (+3)\ 0)$
$\Longrightarrow$ {por $iterate$}
$take\ 2\ (0 \ : \ iterate\ (+3)\ (0+3))$
$\Longrightarrow$ {por $take$}
$0 \ : \ take\ 1\ (iterate\ (+3)\ (0+3))$
$\Longrightarrow$ {por $iterate$ ya que $take$ no puede evaluarse}
$0 \ : \ take\ 1\ ((0+3) \ : \ iterate\ (+3)\ ((0+3)+3))$
$\Longrightarrow$ {por $take$}
$0 \ : \ (0+3) \ : \ take\ 0\ (iterate\ (+3)\ ((0+3)+3))$
$\Longrightarrow$ {por $(+)$}
$0 \ : \ 3 \ : \ take\ 0\ (iterate\ (+3)\ ((0+3)+3))$
$\Longrightarrow$ {por $take$}
$0 \ : \ 3 \ : \ [\,]$
$\Longrightarrow$ {sintaxis de listas}
$[0,\ 3]$

✓ Solo se evalúa la parte de la lista necesaria para el obtener el resultado.

11.2 La criba de Eratóstenes

- ✓ Eratóstenes propuso el un método para calcular todos los números primos
- ✓ Partir de la lista $l_0 = [2 ..]$

$$[\boxed{2}, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, \ldots]$$

- ✓ El primer elemento ($\boxed{2}$) es primo.
- ✓ Calcular l_1 eliminando de l_0 los múltiplos de 2:

l_1

$\Longrightarrow$ {eliminando múltiplos de 2 en l_0}

$[3, \cancel{4}, 5, \cancel{6}, 7, \cancel{8}, 9, \cancel{10}, 11, \cancel{12}, 13, \cancel{14}, 15, \cancel{16}, 17, \cancel{18}, 19, \cancel{20}, 21, \ldots]$

$\Longrightarrow$

$[\boxed{3}, 5, 7, 9, 11, 13, 15, 17, 19, 21, \ldots]$

- ✓ El primer elemento ($\boxed{3}$) es primo.
- ✓ Calcular l_2 eliminando de l_1 los múltiplos de 3:

l_2

$\Longrightarrow$ {eliminando múltiplos de 3 en l_1}

$[5, 7, \cancel{9}, 11, 13, \cancel{15}, 17, 19, \cancel{21}, \ldots]$

$\Longrightarrow$

$[\boxed{5}, 7, 11, 13, 17, 19, \ldots]$

- ✓ Repetir indefinidamente el proceso.
- ✓ Los elementos que aparecen al inicio de las listas $[l_0, l_1, l_2, \ldots]$ son los números primos: [2,3,5,...]

La criba de Eratóstenes (2)

✓ Elimina de una lista el primer elemento y todos sus múltiplos:

```
cribar            :: [Integer] → [Integer]
cribar []         =  []
cribar (x : xs)   =  [ y | y ← xs, y `noEsMúltiploDe` x ]
  where
    a `noEsMúltiploDe` b  =  (mod a b /= 0)
```

Por ejemplo:

```
? cribar [2 .. 10]
[3, 5, 7, 9]  :: [Integer]
```

✓ Observación:

$$
\begin{aligned}
l_0 &\equiv [2\,..\,] \\
l_1 &\equiv cribar\ l_0 \\
l_2 &\equiv cribar\ l_1 \equiv cribar\ (cribar\ l_0) \\
l_3 &\equiv cribar\ l_2 \equiv cribar\ (cribar\ (cribar\ l_0)) \\
&\ldots
\end{aligned}
$$

✓ Una función que devuelva la lista $[\ l_0,\ l_1,\ l_2,\ l_3,\ \ldots\ ]$.

```
cribas  :: [[Integer]]
cribas  =  iterate cribar [2 .. ]
```

✓ El primer primo está al inicio de l_0, el segundo al inicio de l_1, el tercero al inicio de l_2, etc.

✓ La lista infinita de los número primos es:

$$[\ head\ l_0,\ head\ l_1,\ head\ l_2,\ \ldots\ ] \equiv map\ head\ cribas$$

✓ Podemos definir:

```
losPrimos  :: [Integer]
losPrimos  =  map head cribas
```

```
? take 10 losPrimos
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]  :: [Integer]
```

La criba de Eratóstenes (3)

✓ La evaluación perezosa es una técnica potente.

✓ ¿Cuantos primos hay menores que 100?

```
? takeWhile (< 100) losPrimos
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,
61,67,71,73,79,83,89,97] :: [Integer]

? length (takeWhile (< 100) losPrimos)
25 :: Int
```

✓ ¿Cuánto suman?

```
? sum (takeWhile (< 100) losPrimos)
1060 :: Integer
```

✓ ¿Cuál es el primer primo mayor que 100?

```
primeroQue          :: (a → Bool) → [a] → a
primeroQue p []     = error "ninguno cumple la propiedad"
primeroQue p (x : xs)
  | p x             = x
  | otherwise       = primeroQue p xs

? primeroQue (> 100) losPrimos
101 :: Integer
```

✓ ¿Cuál es el primer primo que acaba en 9?

```
acabaEn       :: Integer → Integer → Bool
n `acabaEn` d = (n `mod` 10 == d)

? primeroQue (`acabaEn` 9) losPrimos
19 :: Integer
```

11.3 Redes de procesos

✓ Deseamos calcular úna lista infinita con la sucesión de fibonacci

✓ Podemos definir una función para calcular el i-ésimo número de fibonacci

$$
\begin{array}{lcl}
fib & :: & Integer \to Integer \\
fib\ 0 & = & 0 \\
fib\ 1 & = & 1 \\
fib\ (n+2) & = & fib\ (n+1)\ +\ fib\ n
\end{array}
$$

✓ Usando map podemos construir la secuencia infinita:

$$
\begin{array}{lcl}
fibs & :: & [Integer] \\
fibs & = & map\ fib\ [0\ ..\ ]
\end{array}
$$

Por ejemplo

? $take\ 10\ fibs$
$[0,1,1,2,3,5,8,13,21,34]\ ::\ [Integer]$

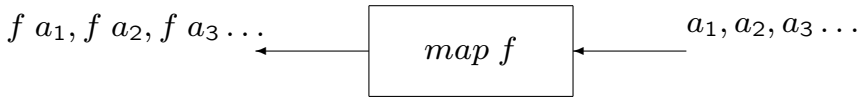
✓ **Problema:** cada término de la sucesión se computa por separado

✓ Podríamos obtener una versión más eficiente si para computar un término solo sumamos los dos previos

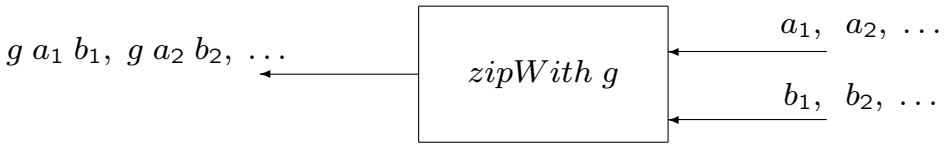
✓ Esto se puede conseguir con una red de procesos

Redes de procesos (2)

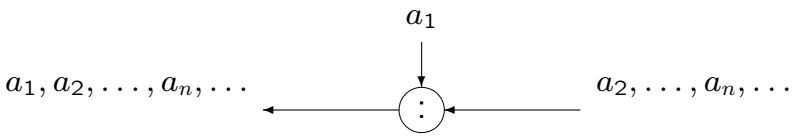
✓ Consideraremos una función $map\ f\ ::\ [a] \rightarrow [b]$ un proceso



✓ Para funciones $zipWith\ g\ ::\ [a] \rightarrow [b] \rightarrow [c]$

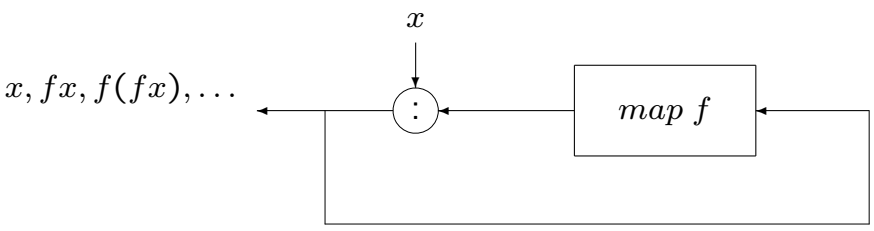


✓ Para el operador $(:)\ ::\ a \rightarrow [a] \rightarrow [a]$



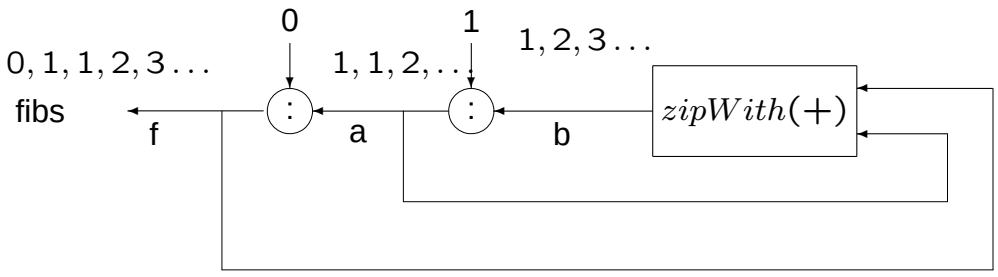
✓ Conectando estos elementos podemos construir redes arbitrarias

✓ Por ejemplo, red que calcula $[\ x,\ f\ x,\ f(f\ x),\ f\ (f\ (f\ x)),\ \ldots\ ]$



Los números de Fibonacci

✓ La red de procesos es



✓ El programa es:

$$
\begin{array}{l}
fibs \;::\; [Integer] \\
fibs \;=\; f \\
\quad \textbf{where} \\
\quad\quad f \;=\; 0 \;:\; a \\
\quad\quad a \;=\; 1 \;:\; b \\
\quad\quad b \;=\; zipWith\ (+)\ f\ a
\end{array}
$$

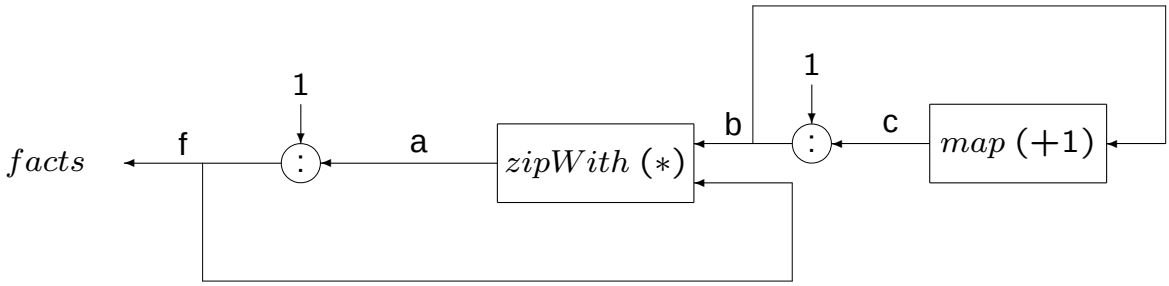
✓ Podemos transformar el programa:

$$
\begin{array}{ll}
 & fibs \;=\; f \textbf{ where } f \;=\; 0 : a;\; a \;=\; 1 : b;\; b \;=\; zipWith\ (+)\ f\ a \\
\equiv & \quad \{\text{por } fibs \;=\; f\} \\
 & fibs \;=\; f \textbf{ where } f \;=\; 0 : a;\; a \;=\; 1 : b;\; b \;=\; zipWith\ (+)\ fibs\ a \\
\equiv & \quad \{\text{por } f \;=\; 0 : a\} \\
 & fibs \;=\; 0 : a \textbf{ where } a \;=\; 1 : b;\; b \;=\; zipWith\ (+)\ fibs\ a \\
\equiv & \quad \{\text{por } a \;=\; 1 : b\} \\
 & fibs \;=\; 0 : 1 : b \textbf{ where } b \;=\; zipWith\ (+)\ fibs\ (1 : b) \\
\equiv & \quad \{\text{ya que } 1 : b \;=\; tail\ fibs\} \\
 & fibs \;=\; 0 : 1 : b \textbf{ where } b \;=\; zipWith\ (+)\ fibs\ (tail\ fibs) \\
\equiv & \quad \{\text{ya que } b \;=\; zipWith\ (+)\ fibs\ (tail\ fibs)\} \\
 & fibs \;=\; 0 : 1 : zipWith\ (+)\ fibs\ (tail\ fibs)
\end{array}
$$

$$
\begin{array}{l}
fibs \;::\; [Integer] \\
fibs \;=\; 0 : 1 : zipWith\ (+)\ fibs\ (tail\ fibs)
\end{array}
$$

Los factoriales

✓ La red de procesos es



✓ El programa es:

```
facts :: [Integer]
facts = f
  where
    f = 1 : a
    a = zipWith (*) b f
    b = 1 : c
    c = map (+1) b
```

✓ Podemos transformar el programa:

```
facts :: [Integer]
facts = 1 : zipWith (*) [1..] facts
```

Objetivos del tema

El alumno debe:

- ✓ Conocer las funciones predefinidas para construir listas infinitas
- ✓ Saber como se evaluan las expresiones usando evaluación perezosa
- ✓ Saber construir redes de procesos para construir listas infinitas
- ✓ Saber construir un programa a partir de una red de procesos