

# Temas de “Programación funcional” (curso 2009–10)

José A. Alonso Jiménez

---

Grupo de Lógica Computacional

Dpto. de Ciencias de la Computación e Inteligencia Artificial

Universidad de Sevilla

Sevilla, 3 de Septiembre de 2009 (versión de 25 de septiembre de 2009)

Esta obra está bajo una licencia Reconocimiento–NoComercial–CompartirIgual 2.5 Spain de Creative Commons.

**Se permite:**

- copiar, distribuir y comunicar públicamente la obra
- hacer obras derivadas

**Bajo las condiciones siguientes:**



**Reconocimiento.** Debe reconocer los créditos de la obra de la manera especificada por el autor.



**No comercial.** No puede utilizar esta obra para fines comerciales.



**Compartir bajo la misma licencia.** Si altera o transforma esta obra, o genera una obra derivada, sólo puede distribuir la obra generada bajo una licencia idéntica a ésta.

- Al reutilizar o distribuir la obra, tiene que dejar bien claro los términos de la licencia de esta obra.
- alguna de estas condiciones puede no aplicarse si se obtiene el permiso del titular de los derechos de autor.

Esto es un resumen del texto legal (la licencia completa). Para ver una copia de esta licencia, visite <http://creativecommons.org/licenses/by-nc-sa/2.5/es/> o envíe una carta a Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

# Índice general

<b>1. Introducción a la programación funcional</b>	<b>5</b>
1.1. Funciones	5
1.2. Programación funcional	7
1.3. Rasgos característicos de Haskell	8
1.4. Antecedentes históricos	9
1.5. Presentación de Haskell	9
<b>2. Introducción a la programación con Haskell</b>	<b>13</b>
2.1. El sistema GHC	13
2.2. Iniciación a GHC	13
2.2.1. Inicio de sesión con GHCi	13
2.2.2. Cálculo aritmético	14
2.2.3. Cálculo con listas	14
2.2.4. Cálculos con errores	15
2.3. Aplicación de funciones	16
2.4. Guiones Haskell	17
2.4.1. El primer guión Haskell	17
2.4.2. Nombres de funciones	18
2.4.3. La regla del sangrado	18
2.4.4. Comentarios en Haskell	19
<b>3. Tipos y clases</b>	<b>21</b>
3.1. Conceptos básicos sobre tipos	21
3.2. Tipos básicos	22
3.3. Tipos compuestos	23
3.3.1. Tipos listas	23
3.3.2. Tipos tuplas	24
3.3.3. Tipos funciones	24
3.4. Parcialización	25
3.5. Polimorfismo y sobrecarga	27
3.5.1. Tipos polimórficos	27

3.5.2. Tipos sobrecargados . . . . .	28
3.6. Clases básicas . . . . .	29
<b>4. Definición de funciones</b>	<b>35</b>
4.1. Definiciones por composición . . . . .	35
4.2. Definiciones con condicionales . . . . .	35
4.3. Definiciones con ecuaciones con guardas . . . . .	36
4.4. Definiciones con equiparación de patrones . . . . .	36
4.4.1. Constantes como patrones . . . . .	36
4.4.2. Variables como patrones . . . . .	37
4.4.3. Tuplas como patrones . . . . .	37
4.4.4. Listas como patrones . . . . .	37
4.4.5. Patrones enteros . . . . .	38
4.5. Expresiones lambda . . . . .	38
4.6. Secciones . . . . .	40
<b>5. Definiciones de listas por comprensión</b>	<b>43</b>
5.1. Generadores . . . . .	43
5.2. Guardas . . . . .	44
5.3. La función zip . . . . .	45
5.4. Comprensión de cadenas . . . . .	46
5.5. Cifrado César . . . . .	47
5.5.1. Codificación y decodificación . . . . .	48
5.5.2. Análisis de frecuencias . . . . .	50
5.5.3. Descifrado . . . . .	51
<b>6. Funciones recursivas</b>	<b>53</b>
6.1. Recursión numérica . . . . .	53
6.2. Recursión sobre lista . . . . .	54
6.3. Recursión sobre varios argumentos . . . . .	57
6.4. Recursión múltiple . . . . .	57
6.5. Recursión mutua . . . . .	58
6.6. Heurísticas para las definiciones recursivas . . . . .	59
<b>7. Razonamiento sobre programas</b>	<b>63</b>
7.1. Razonamiento ecuacional . . . . .	63
7.1.1. Cálculo con longitud . . . . .	63
7.1.2. Propiedad de intercambia . . . . .	63
7.1.3. Inversa de listas unitarias . . . . .	64
7.1.4. Razonamiento ecuacional con análisis de casos . . . . .	65
7.2. Razonamiento por inducción sobre los naturales . . . . .	65

---

7.2.1.	Esquema de inducción sobre los naturales	65
7.2.2.	Ejemplo de inducción sobre los naturales	66
7.3.	Razonamiento por inducción sobre listas	67
7.3.1.	Esquema de inducción sobre listas	67
7.3.2.	Asociatividad de ++	67
7.3.3.	[] es la identidad para ++ por la derecha	68
7.3.4.	Relación entre length y ++	69
7.3.5.	Relación entre take y drop	70
7.3.6.	La concatenación de listas vacías es vacía	71
7.4.	Equivalencia de funciones	72
<b>8.</b>	<b>Funciones de orden superior</b>	<b>75</b>
8.1.	Funciones de orden superior	75
8.2.	Procesamiento de listas	76
8.2.1.	La función map	76
8.2.2.	La función filter	78
8.3.	Función de plegado por la derecha: foldr	79
8.4.	Función de plegado por la izquierda: foldl	82
8.5.	Composición de funciones	83
8.6.	Caso de estudio: Codificación binaria y transmisión de cadenas	84
8.6.1.	Cambio de bases	84
8.6.2.	Codificación y decodificación	86
<b>9.</b>	<b>Declaraciones de tipos y clases</b>	<b>91</b>
9.1.	Declaraciones de tipos	91
9.2.	Definiciones de tipos de datos	93
9.3.	Definición de tipos recursivos	95
9.4.	Sistema de decisión de tautologías	99
9.5.	Máquina abstracta de cálculo aritmético	102
9.6.	Declaraciones de clases y de instancias	104
<b>10.</b>	<b>Evaluación perezosa</b>	<b>109</b>
10.1.	Estrategias de evaluación	109
10.2.	Terminación	110
10.3.	Número de reducciones	111
10.4.	Estructuras infinitas	112
10.5.	Programación modular	113
10.6.	Aplicación estricta	114

<b>11. Analizadores sintácticos funcionales</b>	<b>119</b>
11.1. Analizadores sintácticos	119
11.2. El tipo de los analizadores sintácticos	119
11.3. Analizadores sintácticos básicos	120
11.4. Composición de analizadores sintácticos	121
11.4.1. Secuenciación de analizadores sintácticos	121
11.4.2. Elección de analizadores sintácticos	122
11.5. Primitivas derivadas	122
11.6. Tratamiento de los espacios	125
11.7. Analizador de expresiones aritméticas	126
<b>12. Programas interactivos</b>	<b>133</b>
12.1. Programas interactivos	133
12.2. El tipo de las acciones de entrada/salida	134
12.3. Acciones básicas	134
12.4. Secuenciación	134
12.5. Primitivas derivadas	135
12.6. Ejemplos de programas interactivos	136
12.6.1. Juego de adivinación interactivo	136
12.6.2. Calculadora aritmética	137
12.6.3. El juego de la vida	140
<b>13. Aplicaciones de programación funcional</b>	<b>145</b>
13.1. El juego de cifras y letras	145
13.1.1. Introducción	145
13.1.2. Búsqueda de la solución por fuerza bruta	149
13.1.3. Búsqueda combinando generación y evaluación	151
13.1.4. Búsqueda mejorada mediante propiedades algebraicas	153
13.2. El problema de las reinas	156
13.3. Números de Hamming	157
<b>A. Resumen de funciones predefinidas de Haskell</b>	<b>159</b>
<b>Bibliografía</b>	<b>162</b>

# Tema 1

## Introducción a la programación funcional

### Contenido

---

1.1. Funciones . . . . .	5
1.2. Programación funcional . . . . .	7
1.3. Rasgos característicos de Haskell . . . . .	8
1.4. Antecedentes históricos . . . . .	9
1.5. Presentación de Haskell . . . . .	9

---

### 1.1. Funciones

#### Funciones en Haskell

- En Haskell, una **función** es una **aplicación** que toma uno o más **argumentos** y devuelve un **valor**.
- En Haskell, las funciones se definen mediante **ecuaciones** formadas por el **nombre de la función**, los **nombres de los argumentos** y el **cuerpo** que especifica cómo se calcula el valor a partir de los argumentos.
- Ejemplo de definición de función en Haskell:

```
dobles x = x + x
```

- Ejemplo de evaluación:  
dobles 3  
= 3 + 3 [def. de doble]  
= 6 [def. de +]

## Evaluaciones de funciones en Haskell

- Ejemplo de evaluación anidada impaciente:

```

doble (doble 3)
= doble (3 + 3)    [def. de doble]
= doble 6          [def. de +]
= 6 + 6           [def. de doble]
= 12              [def. de +]

```

- Ejemplo de evaluación anidada perezosa:

```

doble (doble 3)
= (doble 3) + (doble 3) [def. de doble]
= (3 + 3) + (doble 3)  [def. de doble]
= 6 + (doble 3)        [def. de +]
= 6 + (3 + 3)          [def. de doble]
= 6 + 6                [def. de +]
= 12                   [def. de +]

```

## Comprobación de propiedades

- Propiedad: El doble de  $x$  más  $y$  es el doble de  $x$  más el doble de  $y$
- Expresión de la propiedad:

```
prop_doble x y = doble (x+y) == (doble x) + (doble y)
```

- Comprobación de la propiedad con QuickCheck:

```
*Main> quickCheck prop_doble
+++ OK, passed 100 tests.
```

- Para usar QuickCheck hay que importarla, escribiendo al principio del fichero

```
import Test.QuickCheck
```

## Refutación de propiedades

- Propiedad: El producto de dos números cualquiera es distinto de su suma.
- Expresión de la propiedad:

```
prop_prod_suma x y = x*y /= x+y
```



- Refutación de la propiedad con QuickCheck:

```
*Main> quickCheck prop_prod_suma
*** Failed! Falsifiable (after 1 test):
0
0
```

- Refinamiento: El producto de dos números no nulos cualesquiera es distinto de su suma.

```
prop_prod_suma' x y =
  x /= 0 && y /= 0 ==> x*y /= x+y
```

- Refutación de la propiedad con QuickCheck:

```
*Main> quickCheck prop_prod_suma'
+++ OK, passed 100 tests.
*Main> quickCheck prop_prod_suma'
*** Failed! Falsifiable (after 5 tests):
2
2
```

## 1.2. Programación funcional

### Programación funcional y programación imperativa

- La **programación funcional** es un estilo de programación cuyo método básico de computación es la aplicación de funciones a sus argumentos.
- Un **lenguaje de programación funcional** es uno que soporta y potencia el estilo funcional.
- La **programación imperativa** es un estilo de programación en el que los programas están formados por instrucciones que especifican cómo se ha de calcular el resultado.
- Ejemplo de problema para diferenciar los estilos de programación: Sumar los  $n$  primeros números.

### Solución mediante programación imperativa

- Programa *suma n*:
  - contador := 0
  - total := 0
  - repetir**
    - contador := contador + 1
    - total := total + contador
  - hasta que** contador = n

- Evaluación de *suma 4*:

contador	total
0	0
1	1
2	3
3	6
4	10

### Solución mediante programación funcional

- Programa:

```
suma n = sum [1..n]
```

- Evaluación de *suma 4*:
  - suma 4
  - = sum [1..4]      [def. de suma]
  - = sum [1, 2, 3, 4] [def. de [..]]
  - = 1 + 2 + 3 + 4    [def. de sum]
  - = 10              [def. de +]

## 1.3. Rasgos característicos de Haskell

- Programas concisos.
- Sistema potente de tipos.
- Listas por comprensión.
- Funciones recursivas.
- Funciones de orden superior.

- Efectos monádicos.
- Evaluación perezosa.
- Razonamiento sobre programas.

## 1.4. Antecedentes históricos

- 1930s: Alonzo Church desarrolla el lambda cálculo (teoría básica de los lenguajes funcionales).
- 1950s: John McCarthy desarrolla el Lisp (lenguaje funcional con asignaciones).
- 1960s: Peter Landin desarrolla ISWIN (lenguaje funcional puro).
- 1970s: John Backus desarrolla FP (lenguaje funcional con orden superior).
- 1970s: Robin Milner desarrolla ML (lenguaje funcional con tipos polimórficos e inferencia de tipos).
- 1980s: David Turner desarrolla Miranda (lenguaje funcional perezoso).
- 1987: Un comité comienza el desarrollo de Haskell.
- 2003: El comité publica el “Haskell Report”.

## 1.5. Presentación de Haskell

### Ejemplo de recursión sobre listas

- Especificación:  $(\text{sum } xs)$  es la suma de los elementos de  $xs$ .
- Ejemplo:  $\text{sum } [2,3,7] \rightsquigarrow 12$
- Definición:

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

- Evaluación:
  - $\text{sum } [2,3,7]$
  - $= 2 + \text{sum } [3,7]$  [def. de sum]
  - $= 2 + (3 + \text{sum } [7])$  [def. de sum]
  - $= 2 + (3 + (7 + \text{sum } []))$  [def. de sum]
  - $= 2 + (3 + (7 + 0))$  [def. de sum]
  - $= 12$  [def. de +]

- Tipo de sum:  $(\text{Num } a) \Rightarrow [a] \rightarrow a$

### Ejemplo con listas de comprensión

- Especificación:  $(\text{ordena } xs)$  es la lista obtenida ordenando  $xs$  mediante el algoritmo de ordenación rápida.
- Ejemplo:

```
ordena [4,6,2,5,3] ~> [2,3,4,5,6]
ordena "deacb"    ~> "abcde"
```

- Definición:

```
ordena [] = []
ordena (x:xs) =
  (ordena menores) ++ [x] ++ (ordena mayores)
  where menores = [a | a <- xs, a <= x]
        mayores = [b | b <- xs, b > x]
```

- Tipo de ordena:  $(\text{Ord } a) \Rightarrow [a] \rightarrow [a]$

### Evaluación del ejemplo con listas de comprensión

```
ordena [4,6,2,3]
= (ordena [2,3]) ++ [4] ++ (ordena [6])           [def. ordena]
= ((ordena []) ++ [2] ++ (ordena [3])) ++ [4] ++ (ordena [6]) [def. ordena]
= ([] ++ [2] ++ (ordena [3])) ++ [4] ++ (ordena [6]) [def. ordena]
= ([2] ++ (ordena [3])) ++ [4] ++ (ordena [6,5]) [def. ++]
= ([2] ++ ((ordena []) ++ [3] ++ [])) ++ [4] ++ (ordena [6]) [def. ordena]
= ([2] ++ ([] ++ [3] ++ [])) ++ [4] ++ (ordena [6]) [def. ordena]
= ([2] ++ [3]) ++ [4] ++ (ordena [6])           [def. ++]
= [2,3] ++ [4] ++ (ordena [6])                   [def. ++]
= [2,3,4] ++ (ordena [6])                         [def. ++]
= [2,3,4] ++ ((ordena []) ++ [6] ++ (ordena [])) [def. ordena]
= [2,3,4] ++ ((ordena []) ++ [6] ++ (ordena [])) [def. ordena]
= [2,3,4] ++ ([] ++ [6] ++ [])                   [def. ordena]
= [2,3,4,6]                                       [def. ++]
```

## Bibliografía

1. R. Bird. *Introducción a la programación funcional con Haskell*. Prentice Hall, 2000.

- 
- Cap. 1: Conceptos fundamentales.
2. G. Hutton *Programming in Haskell*. Cambridge University Press, 2007.
    - Cap. 1: Introduction.
  3. B. O'Sullivan, D. Stewart y J. Goerzen *Real World Haskell*. O'Reilly, 2008.
    - Cap. 1: Getting Started.
  4. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. *Razonando con Haskell*. Thompson, 2004.
    - Cap. 1: Programación funcional.
  5. S. Thompson. *Haskell: The Craft of Functional Programming*, Second Edition. Addison-Wesley, 1999.
    - Cap. 1: Introducing functional programming.



## Tema 2

# Introducción a la programación con Haskell

### Contenido

---

<b>2.1. El sistema GHC</b>	<b>13</b>
<b>2.2. Iniciación a GHC</b>	<b>13</b>
2.2.1. Inicio de sesión con GHCi	13
2.2.2. Cálculo aritmético	14
2.2.3. Cálculo con listas	14
2.2.4. Cálculos con errores	15
<b>2.3. Aplicación de funciones</b>	<b>16</b>
<b>2.4. Guiones Haskell</b>	<b>17</b>
2.4.1. El primer guión Haskell	17
2.4.2. Nombres de funciones	18
2.4.3. La regla del sangrado	18
2.4.4. Comentarios en Haskell	19

---

## 2.1. El sistema GHC

### El sistema GHC

- Los programas funcionales pueden evaluarse manualmente (como en el tema anterior).
- Los lenguajes funcionales evalúan automáticamente los programas funcionales.

- Haskell es un lenguaje funcional.
- GHC (Glasgow Haskell Compiler) es el intérprete de Haskell que usaremos en el curso.

## 2.2. Iniciación a GHC

### 2.2.1. Inicio de sesión con GHCi

- Inicio mediante `ghci`

```
| I1M> ghci  
| GHCi, version 6.10.3: http://www.haskell.org/ghc/  :? for help  
| Prelude>
```

- La llamada es `Prelude>`
- Indica que ha cargado las definiciones básicas que forman el prelude y el sistema está listo para leer una expresión, evaluarla y escribir su resultado.

### 2.2.2. Cálculo aritmético

#### Cálculo aritmético: Operaciones aritméticas

- Operaciones aritméticas en Haskell:

```
| Prelude> 2+3  
| 5  
| Prelude> 2-3  
| -1  
| Prelude> 2*3  
| 6  
| Prelude> 7 `div` 2  
| 3  
| Prelude> 2^3  
| 8
```

#### Cálculo aritmético: Precedencia y asociatividad

- Precedencia:



```
|Prelude> 2*10^3
2000
|Prelude> 2+3*4
14
```

- Asociatividad:

```
|Prelude> 2^3^4
2417851639229258349412352
|Prelude> 2^(3^4)
2417851639229258349412352
|Prelude> 2-3-4
-5
|Prelude> (2-3)-4
-5
```

### 2.2.3. Cálculo con listas

#### Cálculo con listas: Seleccionar y eliminar

- Seleccionar el primer elemento de una lista no vacía:

```
|head [1,2,3,4,5] ~> 1
```

- Eliminar el primer elemento de una lista no vacía:

```
|tail [1,2,3,4,5] ~> [2,3,4,5]
```

- Seleccionar el  $n$ -ésimo elemento de una lista (empezando en 0):

```
|[1,2,3,4,5] !! 2 ~> 3
```

- Seleccionar los  $n$  primeros elementos de una lista:

```
|take 3 [1,2,3,4,5] ~> [1,2,3]
```

- Eliminar los  $n$  primeros elementos de una lista:

```
|drop 3 [1,2,3,4,5] ~> [4,5]
```

## Cálculo con listas

- Calcular la longitud de una lista:

```
|length [1,2,3,4,5] ~> 5
```

- Calcular la suma de una lista de números:

```
|sum [1,2,3,4,5] ~> 15
```

- Calcular el producto de una lista de números:

```
|product [1,2,3,4,5] ~> 120
```

- Concatenar dos listas:

```
|[1,2,3] ++ [4,5] ~> [1,2,3,4,5]
```

- Invertir una lista:

```
|reverse [1,2,3,4,5] ~> [5,4,3,2,1]
```

### 2.2.4. Cálculos con errores

#### Ejemplos de cálculos con errores

```
Prelude> 1 'div' 0
*** Exception: divide by zero
Prelude> head []
*** Exception: Prelude.head: empty list
Prelude> tail []
*** Exception: Prelude.tail: empty list
Prelude> [2,3] !! 5
*** Exception: Prelude.(!!): index too large
```

## 2.3. Aplicación de funciones

### Aplicación de funciones en matemáticas y en Haskell

- Notación para funciones en matemáticas:
  - En matemáticas, la aplicación de funciones se representa usando paréntesis y la multiplicación usando yuxtaposición o espacios

- Ejemplo:

$$f(a, b) + cd$$

representa la suma del valor de  $f$  aplicado a  $a$  y  $b$  más el producto de  $c$  por  $d$ .

- Notación para funciones en Haskell:

- En Haskell, la aplicación de funciones se representa usando espacios y la multiplicación usando  $*$ .

- Ejemplo:

$$f\ a\ b + c*d$$

representa la suma del valor de  $f$  aplicado a  $a$  y  $b$  más el producto de  $c$  por  $d$ .

### Prioridad de la aplicación de funciones

- En Haskell, la aplicación de funciones tiene mayor prioridad que los restantes operadores. Por ejemplo, la expresión Haskell  $f\ a + b$  representa la expresión matemática  $f(a) + b$ .
- Ejemplos de expresiones Haskell y matemáticas:

Matemáticas	Haskell
$f(x)$	<code>f x</code>
$f(x, y)$	<code>f x y</code>
$f(g(x))$	<code>f (g x)</code>
$f(x, g(y))$	<code>f x (g y)</code>
$f(x)g(y)$	<code>f x * g y</code>

## 2.4. Guiones Haskell

- En Haskell los usuarios pueden definir funciones.
- Las nuevas definiciones se definen en guiones, que son ficheros de textos compuestos por una sucesión de definiciones.
- Se acostumbra a identificar los guiones de Haskell mediante el sufijo `.hs`

### 2.4.1. El primer gui3n Haskell

- Iniciar emacs y abrir dos ventanas: `C-x 2`
- En la primera ventana ejecutar Haskell: `M-x run-haskell`
- Cambiar a la otra ventana: `C-x o`

- Iniciar el guión: `C-x C-f ejemplo.hs`
- Escribir en el guión las siguientes definiciones

```
doble x      = x+x
cuadruple x = doble (doble x)
```

- Grabar el guión: `C-x C-s`
- Cargar el guión en Haskell: `C-c C-l`
- Evaluar ejemplos:

```
*Main> cuadruple 10
40
*Main> take (doble 2) [1,2,3,4,5,6]
[1,2,3,4]
```

- Volver al guión: `C-x o`
- Añadir al guión las siguientes definiciones:

```
factorial n = product [1..n]
media ns    = sum ns `div` length ns
```

- Grabar el guión: `C-x s`
- Cargar el guión en Haskell: `C-c C-l`
- Evaluar ejemplos:

```
*Main> factorial (doble 2)
24
*Main> doble (media [1,5,3])
6
```

### 2.4.2. Nombres de funciones

- Los nombres de funciones tienen que empezar por una letra en minúscula. Por ejemplo,
  - `sumaCuadrado`, `suma_cuadrado`, `suma`
- Las palabras reservadas de Haskell no pueden usarse en los nombres de funciones. Algunas palabras reservadas son

```
case class data default deriving do else
if import in infix infixl infixr instance
let module newtype of then type where
```

- Se acostumbra escribir los argumentos que son listas usando `s` como sufijo de su nombre. Por ejemplo,
  - `ns` representa una lista de números,
  - `xs` representa una lista de elementos,
  - `css` representa una lista de listas de caracteres.

### 2.4.3. La regla del sangrado

- En Haskell la disposición del texto del programa (el **sangrado**) delimita las definiciones mediante la siguiente regla:

Una definición acaba con el primer trozo de código con un margen izquierdo menor o igual que el del comienzo de la definición actual.

- Ejemplo:

```
a = b + c
  where
    b = 1
    c = 2
d = a * 2
```

- Consejos:
  - Comenzar las definiciones de las funciones en la primera columna.
  - Usar el tabulador en emacs para determinar el sangrado en las definiciones.

### 2.4.4. Comentarios en Haskell

- En los guiones Haskell pueden incluirse comentarios.
- Un **comentario simple** comienza con `--` y se extiende hasta el final de la línea.
- Ejemplo de comentario simple:

```
-- (factorial n) es el factorial del número n.
factorial n = product [1..n]
```

- Un **comentario anidado** comienza con {- y termina en -}
- Ejemplo de comentario anidado:

```
{- (factorial n) es el factorial del número n.  
   Por ejemplo, factorial 3 == 6 -}  
factorial n = product [1..n]
```

## Bibliografía

1. R. Bird. *Introducción a la programación funcional con Haskell*. Prentice Hall, 2000.
  - Cap. 1: Conceptos fundamentales.
2. G. Hutton *Programming in Haskell*. Cambridge University Press, 2007.
  - Cap. 2: First steps.
3. B. O'Sullivan, D. Stewart y J. Goerzen *Real World Haskell*. O'Reilly, 2008.
  - Cap. 1: Getting Started.
4. B. Pope y A. van IJzendoorn *A Tour of the Haskell Prelude (basic functions)*
5. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. *Razonando con Haskell*. Thompson, 2004.
  - Cap. 2: Introducción a Haskell.
6. S. Thompson. *Haskell: The Craft of Functional Programming*, Second Edition. Addison-Wesley, 1999.
  - Cap. 2: Getting started with Haskell and Hugs.

# Tema 3

## Tipos y clases

### Contenido

---

<b>3.1. Conceptos básicos sobre tipos</b>	21
<b>3.2. Tipos básicos</b>	22
<b>3.3. Tipos compuestos</b>	23
3.3.1. Tipos listas	23
3.3.2. Tipos tuplas	24
3.3.3. Tipos funciones	24
<b>3.4. Parcialización</b>	25
<b>3.5. Polimorfismo y sobrecarga</b>	27
3.5.1. Tipos polimórficos	27
3.5.2. Tipos sobrecargados	28
<b>3.6. Clases básicas</b>	29

---

### 3.1. Conceptos básicos sobre tipos

#### ¿Qué es un tipo?

- Un **tipo** es una colección de valores relacionados.
- Un ejemplo de tipos es el de los valores booleanos: `Bool`
- El tipo `Bool` tiene dos valores `True` (verdadero) y `False` (falso).
- $v :: T$  representa que  $v$  es un valor del tipo  $T$  y se dice que “ $v$  tiene tipo  $T$ ”.
- Cálculo de tipo con `:type`

```

Prelude> :type True
True :: Bool
Prelude> :type False
False :: Bool

```

- El tipo `Bool -> Bool` está formado por todas las funciones cuyo argumento y valor son booleanos.
- Ejemplo de tipo `Bool -> Bool`

```

Prelude> :type not
not :: Bool -> Bool

```

### Inferencia de tipos

- Regla de inferencia de tipos

$$\frac{f :: A \rightarrow B \quad e :: A}{f e :: B}$$

- Tipos de expresiones:

```

Prelude> :type not True
not True :: Bool
Prelude> :type not False
not False :: Bool
Prelude> :type not (not False)
not (not False) :: Bool

```

- Error de tipo:

```

Prelude> :type not 3
Error: No instance for (Num Bool)
Prelude> :type 1 + False
Error: No instance for (Num Bool)

```

### Ventajas de los tipos

- Los lenguajes en los que la inferencia de tipo precede a la evaluación se denominan de **tipos seguros**.
- Haskell es un lenguaje de tipos seguros.
- En los lenguajes de tipos seguros no ocurren errores de tipos durante la evaluación.



- La inferencia de tipos no elimina todos los errores durante la evaluación. Por ejemplo,

```
Prelude> :type 1 `div` 0
1 `div` 0 :: (Integral t) => t
Prelude> 1 `div` 0
*** Exception: divide by zero
```

## 3.2. Tipos básicos

- Bool (**Valores lógicos**):
  - Sus valores son True y False.
- Char (**Caracteres**):
  - Ejemplos: 'a', 'B', '3', '+'
- String (**Cadena de caracteres**):
  - Ejemplos: "abc", "1 + 2 = 3"
- Int (**Enteros de precisión fija**):
  - Enteros entre  $-2^{31}$  y  $2^{31} - 1$ .
  - Ejemplos: 123, -12
- Integer (**Enteros de precisión arbitraria**):
  - Ejemplos: 1267650600228229401496703205376.
- Float (**Reales de precisión arbitraria**):
  - Ejemplos: 1.2, -23.45, 45e-7
- Double (**Reales de precisión doble**):
  - Ejemplos: 1.2, -23.45, 45e-7

### 3.3. Tipos compuestos

#### 3.3.1. Tipos listas

- Una **lista** es una sucesión de elementos del mismo tipo.
- $[T]$  es el tipo de las listas de elementos de tipo  $T$ .
- Ejemplos de listas:

```
[False, True]  :: [Bool]
['a', 'b', 'd'] :: [Char]
["uno", "tres"] :: [String]
```

- Longitudes:
  - La **longitud** de una lista es el número de elementos.
  - La lista de longitud 0,  $[]$ , es la **lista vacía**.
  - Las listas de longitud 1 se llaman **listas unitarias**.
- Comentarios:
  - El tipo de una lista no informa sobre su longitud:
 

```
['a', 'b']  :: [Char]
['a', 'b', 'c'] :: [Char]
```
  - El tipo de los elementos de una lista puede ser cualquiera:
 

```
[[ 'a', 'b' ], [ 'c' ]] :: [[Char]]
```

#### 3.3.2. Tipos tuplas

- Una **tupla** es una sucesión de elementos.
- $(T_1, T_2, \dots, T_n)$  es el tipo de las  $n$ -tuplas cuya componente  $i$ -ésima es de tipo  $T_i$ .
- Ejemplos de tuplas:

```
(False, True)      :: (Bool, Bool)
(False, 'a', True) :: (Bool, Char, Bool)
```

- Aridades:
  - La **aridad** de una tupla es el número de componentes.
  - La tupla de aridad 0,  $()$ , es la **tupla vacía**.

- No están permitidas las tuplas de longitud 1.
- Comentarios:
  - El tipo de una tupla informa sobre su longitud:
 

```
('a', 'b')      :: (Char, Char)
('a', 'b', 'c') :: (Char, Char, Char)
```
  - El tipo de los elementos de una tupla puede ser cualquiera:
 

```
((('a', 'b'), ['c', 'd'])) :: ((Char, Char), [Char])
```

### 3.3.3. Tipos funciones

#### Tipos funciones

- Una **función** es una aplicación de valores de un tipo en valores de otro tipo.
- $T_1 \rightarrow T_2$  es el tipo de las funciones que aplica valores del tipo  $T_1$  en valores del tipo  $T_2$ .
- Ejemplos de funciones:

```
not      :: Bool -> Bool
isDigit  :: Char -> Bool
```

#### Funciones con múltiples argumentos o valores

- Ejemplo de función con múltiples argumentos:  
suma (x,y) es la suma de x e y. Por ejemplo, suma (2,3) es 5.

```
suma :: (Int, Int) -> Int
suma (x,y) = x+y
```

- Ejemplo de función con múltiples valores:  
deCeroA 5 es la lista de los números desde 0 hasta n. Por ejemplo, deCeroA n es [0,1,2,3,4,5].

```
deCeroA :: Int -> [Int]
deCeroA n = [0..n]
```

- Notas:
  1. En las definiciones se ha escrito la **signatura** de las funciones.
  2. No es obligatorio escribir la signatura de las funciones.
  3. Es conveniente escribir las signatura.

## 3.4. Parcialización

### Parcialización

- Mecanismo de **parcialización** (*currying* en inglés): Las funciones de más de un argumento pueden interpretarse como funciones que toman un argumento y devuelven otra función con un argumento menos.
- Ejemplo de parcialización:

```
suma' :: Int -> (Int -> Int)
suma' x y = x+y
```

`suma'` toma un entero `x` y devuelve la función `suma' x` que toma un entero `y` y devuelve la suma de `x` e `y`. Por ejemplo,

```
*Main> :type suma' 2
suma' 2 :: Int -> Int
*Main> :type suma' 2 3
suma' 2 3 :: Int
```

- Ejemplo de parcialización con tres argumentos:

```
mult :: Int -> (Int -> (Int -> Int))
mult x y z = x*y*z
```

`mult` toma un entero `x` y devuelve la función `mult x` que toma un entero `y` y devuelve la función `mult x y` que toma un entero `z` y devuelve `x*y*z`. Por ejemplo,

```
*Main> :type mult 2
mult 2 :: Int -> (Int -> Int)
*Main> :type mult 2 3
mult 2 3 :: Int -> Int
*Main> :type mult 2 3 7
mult 2 3 7 :: Int
```

### Aplicación parcial

- Las funciones que toman sus argumentos de uno en uno se llaman **currificadas** (*curried* en inglés).
- Las funciones `suma'` y `mult` son currificadas.

- Las funciones currificadas pueden aplicarse parcialmente. Por ejemplo,

```
*Main> (suma' 2) 3
5
```

- Pueden definirse funciones usando aplicaciones parciales. Por ejemplo,

```
suc :: Int -> Int
suc = suma' 1
```

`suc x` es el sucesor de `x`. Por ejemplo, `suc 2` es 3.

### Convenios para reducir paréntesis

- Convenio 1: Las flechas en los tipos se asocia por la derecha. Por ejemplo,
 

```
Int -> Int -> Int -> Int
```

 representa a
 

```
Int -> (Int -> (Int -> Int))
```
- Convenio 2: Las aplicaciones en de funciones se asocia por la derecha. Por ejemplo,
 

```
mult x y z
```

 representa a
 

```
((mult x) y) z
```
- **Nota:** Todas las funciones con múltiples argumentos se definen en forma currificada, salvo que explícitamente se diga que los argumentos tienen que ser tuplas.

## 3.5. Polimorfismo y sobrecarga

### 3.5.1. Tipos polimórficos

- Un tipo es **polimórfico** (“tiene muchas formas”) si contiene una variable de tipo.
- Una función es **polimórfica** si su tipo es polimórfico.
- La función `length` es polimórfica:

- Comprobación:

```
|Prelude> :type length
|length :: [a] -> Int
```

- Significa que para cualquier tipo `a`, `length` toma una lista de elementos de tipo `a` y devuelve un entero.

- `a` es una variable de tipos.
- Las variables de tipos tienen que empezar por minúscula.
- Ejemplos:

```
length [1, 4, 7, 1]           ~> 4
length ["Lunes", "Martes", "Jueves"] ~> 3
length [reverse, tail]      ~> 2
```

### Ejemplos de funciones polimórficas

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

```
fst (1, 'x')           ~> 1
fst (True, "Hoy")     ~> True
```

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

```
head [2,1,4]          ~> 2
head ['b', 'c']       ~> 'b'
```

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

```
take 3 [3,5,7,9,4]    ~> [3,5,7]
take 2 ['l', 'o', 'l', 'a'] ~> "lo"
take 2 "lola"         ~> "lo"
```

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

```
zip [3,5] "lo" ~> [(3, 'l'), (5, 'o')]
```

- `id :: a -> a`

```
id 3           ~> 3
id 'x'        ~> 'x'
```

### 3.5.2. Tipos sobrecargados

- Un tipo está **sobrecargado** si contiene una restricción de clases.
- Una función está **sobrecargada** si su tipo está sobrecargado.
- La función `sum` está sobrecargada:
  - Comprobación:

```
Prelude> :type sum
sum :: (Num a) => [a] -> a
```

- Significa que para cualquier tipo numérico  $a$ , `sumlength` toma una lista de elementos de tipo  $a$  y devuelve un valor de tipo  $a$ .
- `Num a` es una restricción de clases.
- Las restricciones de clases son expresiones de la forma  $C a$ , donde  $C$  es el nombre de una clase y  $a$  es una variable de tipo.
- Ejemplos:

```
sum [2, 3, 5]           ~> 10
sum [2.1, 3.23, 5.345] ~> 10.675
```

### Ejemplos de tipos sobrecargados

- Ejemplos de funciones sobrecargadas:
  - `(-)` :: (Num a) => a -> a -> a
  - `(*)` :: (Num a) => a -> a -> a
  - `negate` :: (Num a) => a -> a
  - `abs` :: (Num a) => a -> a
  - `signum` :: (Num a) => a -> a
- Ejemplos de números sobrecargados:
  - `5` :: (Num t) => t
  - `5.2` :: (Fractional t) => t

## 3.6. Clases básicas

- Una **clase** es una colección de tipos junto con ciertas operaciones sobrecargadas llamadas **métodos**.
- Clases básicas:

<code>Eq</code>	tipos comparables por igualdad
<code>Ord</code>	tipos ordenados
<code>Show</code>	tipos mostrables
<code>Read</code>	tipos legibles
<code>Num</code>	tipos numéricos
<code>Integral</code>	tipos enteros
<code>Fractional</code>	tipos fraccionarios

### La clase Eq (tipos comparables por igualdad)

- Eq contiene los tipos cuyos valores son comparables por igualdad.

- Métodos:

```
(==) :: a -> a -> Bool
(/=) :: a -> a -> Bool
```

- Instancias:

- Bool, Char, String, Int, Integer, Float y Double.
- tipos compuestos: listas y tuplas.

- Ejemplos:

```
False == True      ~> False
False /= True      ~> True
'a' == 'b'         ~> False
"aei" == "aei"     ~> True
[2,3] == [2,3,2]   ~> False
('a',5) == ('a',5) ~> True
```

### La clase Ord (tipos ordenados)

- Ord es la subclase de Eq de tipos cuyos valores están ordenados.

- Métodos:

```
(<), (<=), (>), (>=) :: a -> a -> Bool
min, man                :: a -> a -> a
```

- Instancias:

- Bool, Char, String, Int, Integer, Float y Double.
- tipos compuestos: listas y tuplas.

- Ejemplos:

```
False < True      ~> True
min 'a' 'b'       ~> 'a'
"elegante" < "elefante" ~> False
[1,2,3] < [1,2]   ~> False
('a',2) < ('a',1) ~> False
('a',2) < ('b',1) ~> True
```



### La clase Show (tipos mostrables)

- Show contiene los tipos cuyos valores se pueden convertir en cadenas de caracteres.

- Método:

```
| show :: a -> String
```

- Instancias:

- Bool, Char, String, Int, Integer, Float y Double.
- tipos compuestos: listas y tuplas.

- Ejemplos:

```
| show False      ~> "False"
| show 'a'        ~> "'a'"
| show 123        ~> "123"
| show [1,2,3]    ~> "[1,2,3]"
| show ('a',True) ~> "('a',True)"
```

### La clase Read (tipos legibles)

- Read contiene los tipos cuyos valores se pueden obtener a partir de cadenas de caracteres.

- Método:

```
| read :: String -> a
```

- Instancias:

- Bool, Char, String, Int, Integer, Float y Double.
- tipos compuestos: listas y tuplas.

- Ejemplos:

```
| read "False" :: Bool      ~> False
| read "'a'"   :: Char      ~> 'a'
| read "123"   :: Int       ~> 123
| read "[1,2,3]" :: [Int]    ~> [1,2,3]
| read "('a',True)" :: (Char,Bool) ~> ('a',True)
```

### La clase Num (tipos numéricos)

- Num es la subclase de Eq y Ord de tipos cuyos valores son números

- Métodos:

```
(+), (*), (-)      :: a -> a -> a
negate, abs, signum :: a -> a
```

- Instancias: Int, Integer, Float y Double.

- Ejemplos:

```
2+3      ~> 5
2.3+4.2  ~> 6.5
negate 2.7 ~> -2.7
abs (-5) ~> 5
signum (-5) ~> -1
```

### La clase Integral (tipos enteros)

- Integral es la subclase de Num cuyo tipos tienen valores enteros.

- Métodos:

```
div :: a -> a -> a
mod :: a -> a -> a
```

- Instancias: Int e Integer.

- Ejemplos:

```
11 `div` 4 ~> 2
11 `mod` 4 ~> 3
```

### La clase Fractional (tipos fraccionarios)

- Fractional es la subclase de Num cuyo tipos tienen valores no son enteros.

- Métodos:

```
(/)  :: a -> a -> a
recip :: a -> a
```

- Instancias: Float y Double.

- Ejemplos:

7.0 / 2.0	↪	3.5
recip 0.2	↪	5.0

## Bibliografía

1. R. Bird. *Introducción a la programación funcional con Haskell*. Prentice Hall, 2000.
  - Cap. 2: Tipos de datos simples.
2. G. Hutton *Programming in Haskell*. Cambridge University Press, 2007.
  - Cap. 3: Types and classes.
3. B. O'Sullivan, D. Stewart y J. Goerzen *Real World Haskell*. O'Reilly, 2008.
  - Cap. 2: Types and Functions.
4. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. *Razonando con Haskell*. Thomson, 2004.
  - Cap. 2: Introducción a Haskell.
  - Cap. 5: El sistema de clases de Haskell.
5. S. Thompson. *Haskell: The Craft of Functional Programming*, Second Edition. Addison-Wesley, 1999.
  - Cap. 3: Basic types and definitions.



# Tema 4

## Definición de funciones

### Contenido

---

<b>4.1. Definiciones por composición</b>	35
<b>4.2. Definiciones con condicionales</b>	35
<b>4.3. Definiciones con ecuaciones con guardas</b>	36
<b>4.4. Definiciones con equiparación de patrones</b>	36
4.4.1. Constantes como patrones	36
4.4.2. Variables como patrones	37
4.4.3. Tuplas como patrones	37
4.4.4. Listas como patrones	37
4.4.5. Patrones enteros	38
<b>4.5. Expresiones lambda</b>	38
<b>4.6. Secciones</b>	40

---

### 4.1. Definiciones por composición

- Decidir si un carácter es un dígito:

```
_____ Prelude _____  
isDigit :: Char -> Bool  
isDigit c = c >= '0' && c <= '9'
```

- Decidir si un entero es par:

```
_____ Prelude _____  
even :: (Integral a) => a -> Bool  
even n = n `rem` 2 == 0
```

- Dividir una lista en su  $n$ -ésimo elemento:

```

Prelude
splitAt :: Int -> [a] -> ([a],[a])
splitAt n xs = (take n xs, drop n xs)

```

## 4.2. Definiciones con condicionales

- Calcular el valor absoluto (con condicionales):

```

Prelude
abs :: Int -> Int
abs n = if n >= 0 then n else -n

```

- Calcular el signo de un número (con condicionales anidados):

```

Prelude
signum :: Int -> Int
signum n = if n < 0 then (-1) else
           if n == 0 then 0 else 1

```

## 4.3. Definiciones con ecuaciones con guardas

- Calcular el valor absoluto (con ecuaciones guardadas):

```

Prelude
abs n | n >= 0    = n
      | otherwise = -n

```

- Calcular el signo de un número (con ecuaciones guardadas):

```

Prelude
signum n | n < 0    = -1
         | n == 0   = 0
         | otherwise = 1

```

## 4.4. Definiciones con equiparación de patrones

### 4.4.1. Constantes como patrones

- Calcular la negación:

---

```
_____ Prelude _____
```

```
not :: Bool -> Bool
not True = False
not False = True
```

- Calcular la conjunción (con valores):

```
_____ Prelude _____
```

```
(&&) :: Bool -> Bool -> Bool
True && True = True
True && False = False
False && True = False
False && False = False
```

#### 4.4.2. Variables como patrones

- Calcular la conjunción (con variables anónimas):

```
_____ Prelude _____
```

```
(&&) :: Bool -> Bool -> Bool
True && True = True
_ && _ = False
```

- Calcular la conjunción (con variables):

```
_____ Prelude _____
```

```
(&&) :: Bool -> Bool -> Bool
True && x = x
False && _ = False
```

#### 4.4.3. Tuplas como patrones

- Calcular el primer elemento de un par:

```
_____ Prelude _____
```

```
fst :: (a,b) -> a
fst (x,_) = x
```

- Calcular el segundo elemento de un par:

```
_____ Prelude _____
```

```
snd :: (a,b) -> b
snd (_,y) = y
```

#### 4.4.4. Listas como patrones

- `(test1 xs)` se verifica si `xs` es una lista de 3 caracteres que empieza por 'a'.

```
test1 :: [Char ] -> Bool
test1 ['a',_,_] = True
test1 _          = False
```

- Construcción de listas con `(:)`

```
[1,2,3] = 1:[2,3] = 1:(2:[3]) = 1:(2:(3:[]))
```

- `(test2 xs)` se verifica si `xs` es una lista de caracteres que empieza por 'a'.

```
test2 :: [Char ] -> Bool
test2 ('a':_) = True
test2 _       = False
```

- Decidir si una lista es vacía:

```
null :: [a] -> Bool
null []      = True
null (_:_)  = False
```

\_\_\_\_\_ Prelude \_\_\_\_\_

- Primer elemento de una lista:

```
head :: [a] -> a
head (x:_) = x
```

\_\_\_\_\_ Prelude \_\_\_\_\_

- Resto de una lista:

```
tail :: [a] -> [a]
tail (_:xs) = xs
```

\_\_\_\_\_ Prelude \_\_\_\_\_

#### 4.4.5. Patrones enteros

- Predecesor de un número entero:

```
pred :: Int -> Int
pred 0      = 0
pred (n+1) = n
```

\_\_\_\_\_ Prelude \_\_\_\_\_



- Comentarios sobre los patrones  $n+k$ :
  - $n+k$  sólo se equipara con números mayores o iguales que  $k$
  - Hay que escribirlo entre paréntesis.

## 4.5. Expresiones lambda

- Las funciones pueden construirse sin nombrarlas mediante las expresiones lambda.
- Ejemplo de evaluación de expresiones lambda:

```
Prelude> (\x -> x+x) 3
6
```

Uso de las expresiones lambda para resaltar la parcialización:

- `(suma x y)` es la suma de  $x$  e  $y$ .
- Definición sin lambda:

```
suma x y = x+y
```

- Definición con lambda:

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

Uso de las expresiones lambda en funciones como resultados:

- `(const x y)` es  $x$ .
- Definición sin lambda:

```

Prelude
const :: a -> b -> a
const x = x
```

- Definición con lambda:

```
const' :: a -> (b -> a)
const' x = \_ -> x
```

Uso de las expresiones lambda en funciones con sólo un uso:

- `(impares n)` es la lista de los  $n$  primeros números impares.

- Definición sin lambda:

```
impares n = map f [0..n-1]
  where f x = 2*x+1
```

- Definición con lambda:

```
impares' n = map (\x -> 2*x+1) [0..n-1]
```

## 4.6. Secciones

- Los **operadores** son las funciones que se escriben entre sus argumentos.
- Los operadores pueden convertirse en funciones prefijas escribiéndolos entre paréntesis.
- Ejemplo de conversión:

```
Prelude> 2 + 3
5
Prelude> (+) 2 3
5
```

- Ejemplos de secciones:

```
Prelude> (2+) 3
5
Prelude> (+3) 2
5
```

### Expresión de secciones mediante lambdas

Sea \* un operador. Entonces

- (\*) = \x -> (\y -> x\*y)
- (x\*) = \y -> x\*y
- (\*y) = \x -> x\*y

## Aplicaciones de secciones

- Uso en definiciones de funciones mediante secciones

```
suma'      = (+)
siguiente  = (1+)
inverso    = (1/)
doble      = (2*)
mitad      = (/2)
```

- Uso en signatura de operadores:

```
_____ Prelude _____
(&&) :: Bool -> Bool -> Bool
```

- Uso como argumento:

```
| Prelude> map (2*) [1..5]
| [2,4,6,8,10]
```

## Bibliografía

1. R. Bird. *Introducción a la programación funcional con Haskell*. Prentice Hall, 2000.
  - Cap. 1: Conceptos fundamentales.
2. G. Hutton *Programming in Haskell*. Cambridge University Press, 2007.
  - Cap. 4: Defining functions.
3. B. O'Sullivan, D. Stewart y J. Goerzen *Real World Haskell*. O'Reilly, 2008.
  - Cap. 2: Types and Functions.
4. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. *Razonando con Haskell*. Thomson, 2004.
  - Cap. 2: Introducción a Haskell.
5. S. Thompson. *Haskell: The Craft of Functional Programming*, Second Edition. Addison-Wesley, 1999.
  - Cap. 3: Basic types and definitions.



# Tema 5

## Definiciones de listas por comprensión

### Contenido

---

5.1. Generadores . . . . .	43
5.2. Guardas . . . . .	44
5.3. La función zip . . . . .	45
5.4. Comprensión de cadenas . . . . .	46
5.5. Cifrado César . . . . .	47
5.5.1. Codificación y decodificación . . . . .	48
5.5.2. Análisis de frecuencias . . . . .	50
5.5.3. Descifrado . . . . .	51

---

### 5.1. Generadores

#### Definiciones por comprensión

- Definiciones por comprensión en Matemáticas:

$$\{x^2 : x \in \{2,3,4,5\}\} = \{4,9,16,25\}$$

- Definiciones por comprensión en Haskell:

```
|Prelude> [x^2 | x <- [2..5]]  
[4,9,16,25]
```

- La expresión `x <- [2..5]` se llama un **generador**.
- Ejemplos con más de un generador:

```

Prelude> [(x,y) | x <- [1,2,3], y <- [4,5]]
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
Prelude> [(x,y) | y <- [4,5], x <- [1,2,3]]
[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]

```

## Generadores dependientes

- Ejemplo con generadores dependientes:

```

Prelude> [(x,y) | x <- [1..3], y <- [x..3]]
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]

```

- (`concat xss`) es la concatenación de la lista de listas `xss`. Por ejemplo,

```

concat [[1,3],[2,5,6],[4,7]] ~> [1,3,2,5,6,4,7]

```

```

----- Prelude -----
concat :: [[a]] -> [a]
concat xss = [x | xs <- xss, x <- xs]

```

## Generadores con variables anónimas

- Ejemplo de generador con variable anónima:  
(`primeros ps`) es la lista de los primeros elementos de la lista de pares `ps`. Por ejemplo,

```

primeros [(1,3),(2,5),(6,3)] ~> [1,2,6]

```

```

primeros :: [(a, b)] -> [a]
primeros ps = [x | (x,_) <- ps]

```

- Definición de la longitud por comprensión

```

----- Prelude -----
length :: [a] -> Int
length xs = sum [1 | _ <- xs]

```

## 5.2. Guardas

- Las listas por comprensión pueden tener **guardas** para restringir los valores.
- Ejemplo de guarda:

```
Prelude> [x | x <- [1..10], even x]
[2,4,6,8,10]
```

La guarda es `even x`.

- `(factores n)` es la lista de los factores del número `n`. Por ejemplo,

```
factores 30 ~> [1,2,3,5,6,10,15,30]
```

```
factores :: Int -> [Int]
factores n = [x | x <- [1..n], n `mod` x == 0]
```

- `(primo n)` se verifica si `n` es primo. Por ejemplo,

```
primo 30 ~> False
primo 31 ~> True
```

```
primo :: Int -> Bool
primo n = factores n == [1, n]
```

- `(primos n)` es la lista de los primos menores o iguales que `n`. Por ejemplo,

```
primos 31 ~> [2,3,5,7,11,13,17,19,23,29,31]
```

```
primos :: Int -> [Int]
primos n = [x | x <- [2..n], primo x]
```

### Guarda con igualdad

- Una **lista de asociación** es una lista de pares formado por una clave y un valor. Por ejemplo,

```
[("Juan",7),("Ana",9),("Eva",3)]
```

- `(busca c t)` es la lista de los valores de la lista de asociación `t` cuyas claves valen `c`. Por ejemplo,

```
Prelude> busca 'b' [('a',1),('b',3),('c',5),('b',2)]
[3,2]
```

```
busca :: Eq a => a -> [(a, b)] -> [b]
busca c t = [v | (c', v) <- t, c' == c]
```

### 5.3. La función zip

#### La función zip y elementos adyacentes

- (zip xs ys) es la lista obtenida emparejando los elementos de las listas xs e ys. Por ejemplo,

```
Prelude> zip ['a','b','c'] [2,5,4,7]
 [('a',2),('b',5),('c',4)]
```

- (adyacentes xs) es la lista de los pares de elementos adyacentes de la lista xs. Por ejemplo,

```
adyacentes [2,5,3,7] ~> [(2,5),(5,3),(3,7)]
```

```
adyacentes :: [a] -> [(a, a)]
adyacentes xs = zip xs (tail xs)
```

#### Las funciones zip, and y listas ordenadas

- (and xs) se verifica si todos los elementos de xs son verdaderos. Por ejemplo,

```
and [2 < 3, 2+3 == 5] ~> True
and [2 < 3, 2+3 == 5, 7 < 7] ~> False
```

- (ordenada xs) se verifica si la lista xs está ordenada. Por ejemplo,

```
ordenada [1,3,5,6,7] ~> True
ordenada [1,3,6,5,7] ~> False
```

```
ordenada :: Ord a => [a] -> Bool
ordenada xs = and [x <= y | (x,y) <- adyacentes xs]
```



### La función zip y lista de posiciones

- (posiciones x xs) es la lista de las posiciones ocupadas por el elemento x en la lista xs. Por ejemplo,

```
|posiciones 5 [1,5,3,5,5,7] ~> [1,3,4]
```

```
posiciones :: Eq a => a -> [a] -> [Int]
posiciones x xs =
  [i | (x',i) <- zip xs [0..n], x == x']
  where n = length xs - 1
```

## 5.4. Comprensión de cadenas

### Cadenas y listas

- Las cadenas son listas de caracteres. Por ejemplo,

```
|*Main> "abc" == ['a','b','c']
True
```

- La expresión

```
|"abc" :: String
```

es equivalente a

```
|['a','b','c'] :: [Char]
```

- Las funciones sobre listas se aplican a las cadenas:

```
length "abcde"           ~> 5
reverse "abcde"         ~> "edcba"
"abcde" ++ "fg"         ~> "abcdefg"
posiciones 'a' "Salamanca" ~> [1,3,5,8]
```

### Definiciones sobre cadenas con comprensión

- (minusculas c) es la cadena formada por las letras minúsculas de la cadena c. Por ejemplo,

```
|minusculas "EstoEsUnaPrueba" ~> "stosnarueba"
```

```

minuscultas :: String -> String
minuscultas xs = [x | x <- xs, elem x ['a'..'z']]

```

- (ocurrencias x xs) es el número de veces que ocurre el carácter x en la cadena xs. Por ejemplo,

```
|ocurrencias 'a' "Salamanca" ~> 4
```

```

ocurrencias :: Char -> String -> Int
ocurrencias x xs = length [x' | x' <- xs, x == x']

```

## 5.5. Cifrado César

- En el [cifrado César](#) cada letra en el texto original es reemplazada por otra letra que se encuentra 3 posiciones más adelante en el alfabeto.

- La codificación de

```
|"en todo la medida"
```

es

```
|"hq wrgr od phlgd"
```

- Se puede generalizar desplazando cada letra  $n$  posiciones.
- La codificación con un desplazamiento 5 de

```
|"en todo la medida"
```

es

```
|"js ytit qf rjinif"
```

- La descodificación de un texto codificado con un desplazamiento  $n$  se obtiene codificándolo con un desplazamiento  $-n$ .

### 5.5.1. Codificación y descodificación

#### Las funciones `ord` y `char`

- `(ord c)` es el código del carácter `c`. Por ejemplo,

```
ord 'a'  ~> 97
ord 'b'  ~> 98
ord 'A'  ~> 65
```

- `(chr n)` es el carácter de código `n`. Por ejemplo,

```
chr 97  ~> 'a'
chr 98  ~> 'b'
chr 65  ~> 'A'
```

#### Codificación y descodificación: Código de letra

- Simplificación: Sólo se codificarán las letras minúsculas dejando los restantes caracteres sin modificar.
- `(let2int c)` es el entero correspondiente a la letra minúscula `c`. Por ejemplo,

```
let2int 'a' ~> 0
let2int 'd' ~> 3
let2int 'z' ~> 25
```

```
let2int :: Char -> Int
let2int c = ord c - ord 'a'
```

#### Codificación y descodificación: Letra de código

- `(int2let n)` es la letra minúscula correspondiente al entero `n`. Por ejemplo,

```
int2let 0  ~> 'a'
int2let 3  ~> 'd'
int2let 25 ~> 'z'
```

```
int2let :: Int -> Char
int2let n = chr (ord 'a' + n)
```

## Codificación y descodificación: Desplazamiento

- `(desplaza n c)` es el carácter obtenido desplazando  $n$  caracteres el carácter  $c$ . Por ejemplo,

```
desplaza 3 'a' ~> 'd'
desplaza 3 'y' ~> 'b'
desplaza (-3) 'd' ~> 'a'
desplaza (-3) 'b' ~> 'y'
```

```
desplaza :: Int -> Char -> Char
desplaza n c
  | elem c ['a'..'z'] = int2let ((let2int c+n) `mod` 26)
  | otherwise        = c
```

## Codificación y descodificación

- `(codifica n xs)` es el resultado de codificar el texto  $xs$  con un desplazamiento  $n$ . Por ejemplo,

```
Prelude> codifica 3 "En todo la medida"
"Eq wrgr od phglgd"
Prelude> codifica (-3) "Eq wrgr od phglgd"
"En todo la medida"
```

```
codifica :: Int -> String -> String
codifica n xs = [desplaza n x | x <- xs]
```

## Propiedades de la codificación con QuickCheck

- Propiedad: Al desplazar  $-n$  un carácter desplazado  $n$ , se obtiene el carácter inicial.

```
prop_desplaza n xs =
  desplaza (-n) (desplaza n xs) == xs
```

```
*Main> quickCheck prop_desplaza
+++ OK, passed 100 tests.
```

- Propiedad: Al codificar con  $-n$  una cadena codificada con  $n$ , se obtiene la cadena inicial.

```
prop_codifica n xs =
  codifica (-n) (codifica n xs) == xs
```

```
*Main> quickCheck prop_codifica
+++ OK, passed 100 tests.
```

## 5.5.2. Análisis de frecuencias

### Tabla de frecuencias

- Para descifrar mensajes se parte de la [frecuencia de aparición de letras](#).
- `tabla` es la lista de la frecuencias de las letras en castellano, Por ejemplo, la frecuencia de la 'a' es del 12.53%, la de la 'b' es 1.42%.

```
tabla :: [Float]
tabla = [12.53, 1.42, 4.68, 5.86, 13.68, 0.69, 1.01,
         0.70, 6.25, 0.44, 0.01, 4.97, 3.15, 6.71,
         8.68, 2.51, 0.88, 6.87, 7.98, 4.63, 3.93,
         0.90, 0.02, 0.22, 0.90, 0.52]
```

### Frecuencias

- `(porcentaje n m)` es el porcentaje de `n` sobre `m`. Por ejemplo,

```
| porcentaje 2 5 ~> 40.0
```

```
porcentaje :: Int -> Int -> Float
porcentaje n m = (fromIntegral n / fromIntegral m) * 100
```

- `(frecuencias xs)` es la frecuencia de cada una de las minúsculas de la cadena `xs`. Por ejemplo,

```
| Prelude> frecuencias "en todo la medida"
[14.3,0,0,21.4,14.3,0,0,0,7.1,0,0,7.1,
 7.1,7.1,14.3,0,0,0,0,7.1,0,0,0,0,0,0]
```

```
frecuencias :: String -> [Float]
frecuencias xs =
  [porcentaje (ocurrencias x xs) n | x <- ['a'..'z']]
  where n = length (minusculas xs)
```

### 5.5.3. Descifrado

#### Descifrado: Ajuste chi cuadrado

- Una medida de la discrepancia entre la distribución observada  $os_i$  y la esperada  $es_i$  es

$$\chi^2 = \sum_{i=0}^{n-1} \frac{(os_i - es_i)^2}{es_i}$$

Los menores valores corresponden a menores discrepancias.

- `(chiCud os es)` es la medida chi cuadrado de las distribuciones `os` y `es`. Por ejemplo,

```
chiCud [3,5,6] [3,5,6] ~> 0.0
chiCud [3,5,6] [5,6,3] ~> 3.9666667
```

```
chiCud :: [Float] -> [Float] -> Float
chiCud os es =
  sum [((o-e)^2)/e | (o,e) <- zip os es]
```

#### Descifrado: Rotación

- `(rota n xs)` es la lista obtenida rotando `n` posiciones los elementos de la lista `xs`. Por ejemplo,

```
rota 2 "manolo" ~> "noloma"
```

```
rota :: Int -> [a] -> [a]
rota n xs = drop n xs ++ take n xs
```

#### Descifrado

- `(descifra xs)` es la cadena obtenida descodificando la cadena `xs` por el anti-desplazamiento que produce una distribución de minúsculas con la menor desviación chi cuadrado respecto de la tabla de distribución de las letras en castellano. Por ejemplo,

```
*Main> codifica 5 "Todo para nada"
"Ttit ufwf sfif"
*Main> descifra "Ttit ufwf sfif"
"Todo para nada"
```

```
descifra :: String -> String
descifra xs = codifica (-factor) xs
  where
    factor = head (posiciones (minimum tabChi) tabChi)
    tabChi = [chiCuad (rota n tabla') tabla | n <- [0..25]]
    tabla' = frecuencias xs
```

## Bibliografía

1. R. Bird. *Introducción a la programación funcional con Haskell*. Prentice Hall, 2000.
  - Cap. 4: Listas.
2. G. Hutton *Programming in Haskell*. Cambridge University Press.
  - Cap. 5: List comprehensions.
3. B. O'Sullivan, D. Stewart y J. Goerzen *Real World Haskell*. O'Reilly, 2008.
  - Cap. 12: Barcode Recognition.
4. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. *Razonando con Haskell*. Thomson, 2004.
  - Cap. 6: Programación con listas.
5. S. Thompson. *Haskell: The Craft of Functional Programming*, Second Edition. Addison-Wesley, 1999.
  - Cap. 5: Data types: tuples and lists.





# Tema 6

## Funciones recursivas

### Contenido

---

<b>6.1. Recursión numérica</b> . . . . .	53
<b>6.2. Recusión sobre lista</b> . . . . .	54
<b>6.3. Recursión sobre varios argumentos</b> . . . . .	57
<b>6.4. Recursión múltiple</b> . . . . .	57
<b>6.5. Recursión mutua</b> . . . . .	58
<b>6.6. Heurísticas para las definiciones recursivas</b> . . . . .	59

---

### 6.1. Recursión numérica

#### Recursión numérica: El factorial

- La función factorial:

```
factorial :: Integer -> Integer
factorial 0      = 1
factorial (n + 1) = (n + 1) * factorial n
```

- Cálculo:

```
factorial 3 = 3 * (factorial 2)
            = 3 * (2 * (factorial 1))
            = 3 * (2 * (1 * (factorial 0)))
            = 3 * (2 * (1 * 1))
            = 3 * (2 * 1)
            = 3 * 2
            = 6
```

## Recursión numérica: El producto

- Definición recursiva del producto:

```
por :: Int -> Int -> Int
m 'por' 0      = 0
m 'por' (n + 1) = m + (m 'por' n)
```

- Cálculo:

```
3 'por' 2 = 3 + (3 'por' 1)
          = 3 + (3 + (3 'por' 0))
          = 3 + (3 + 0)
          = 3 + 3
          = 6
```

## 6.2. Recusión sobre lista

### Recusión sobre listas: La función product

- Producto de una lista de números:

```
product :: Num a => [a] -> a
product []      = 1
product (n:ns) = n * product ns
```

- Cálculo:

```
product [7,5,2] = 7 * (product [5,2])
               = 7 * (5 * (product [2]))
               = 7 * (5 * (2 * (product [])))
               = 7 * (5 * (2 * 1))
               = 7 * (5 * 2)
               = 7 * 10
               = 70
```

### Recusión sobre listas: La función length

- Longitud de una lista:

Prelude

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

- Cálculo:

```
length [2,3,5] = 1 + (length [3,5])
               = 1 + (1 + (length [5]))
               = 1 + (1 + (1 + (length [])))
               = 1 + (1 + (1 + 0))
               = 1 + (1 + 1)
               = 1 + 2
               = 3
```

### Recursión sobre listas: La función reverse

- Inversa de una lista:

Prelude

```
reverse :: [a] -> [a]
reverse []      = []
reverse (x:xs) = reverse xs ++ [x]
```

- Cálculo:

```
reverse [2,5,3] = (reverse [5,3]) ++ [2]
                = ((reverse [3]) ++ [5]) ++ [2]
                = (((reverse []) ++ [3]) ++ [5]) ++ [2]
                = (([] ++ [3]) ++ [5]) ++ [2]
                = ([3] ++ [5]) ++ [2]
                = [3,5] ++ [2]
                = [3,5,2]
```

### Recursión sobre listas: ++

- Concatenación de listas:

Prelude

```
(++) :: [a] -> [a] -> [a]
[]    ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

- Cálculo:

```

[1,3,5] ++ [2,4] = 1:([3,5] ++ [2,4])
                 = 1:(3:([5] ++ [2,4]))
                 = 1:(3:(5:([ ] ++ [2,4])))
                 = 1:(3:(5:[2,4]))
                 = 1:(3:[5,2,4])
                 = 1:[3,5,2,4]
                 = [1,3,5,2,4]

```

### Recursión sobre listas: Inserción ordenada

- (`inserta e xs`) inserta el elemento `e` en la lista `xs` delante del primer elemento de `xs` mayor o igual que `e`. Por ejemplo,

```
inserta 5 [2,4,7,3,6,8,10] ~> [2,4,5,7,3,6,8,10]
```

```

inserta :: Ord a => a -> [a] -> [a]
inserta e []                = [e]
inserta e (x:xs) | e <= x  = e : (x:xs)
                  | otherwise = x : inserta e xs

```

- Cálculo:

```

inserta 4 [1,3,5,7] = 1:(inserta 4 [3,5,7])
                   = 1:(3:(inserta 4 [5,7]))
                   = 1:(3:(4:(5:[7])))
                   = 1:(3:(4:[5,7]))
                   = [1,3,4,5,7]

```

### Recursión sobre listas: Ordenación por inserción

- (`ordena_por_insercion xs`) es la lista `xs` ordenada mediante inserción, Por ejemplo,

```
ordena_por_insercion [2,4,3,6,3] ~> [2,3,3,4,6]
```

```

ordena_por_insercion :: Ord a => [a] -> [a]
ordena_por_insercion []      = []
ordena_por_insercion (x:xs) =
  inserta x (ordena_por_insercion xs)

```

- Cálculo:

```
ordena_por_insercion [7,9,6] =
= inserta 7 (inserta 9 (inserta 6 []))
= inserta 7 (inserta 9 [6])
= inserta 7 [6,9]
= [6,7,9]
```

### 6.3. Recursión sobre varios argumentos

#### Recursión sobre varios argumentos: La función zip

- Emparejamiento de elementos:

```
----- Prelude -----
zip :: [a] -> [b] -> [(a, b)]
zip [] _ = []
zip _ [] = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

- Cálculo:

```
zip [1,3,5] [2,4,6,8]
= (1,2) : (zip [3,5] [4,6,8])
= (1,2) : ((3,4) : (zip [5] [6,8]))
= (1,2) : ((3,4) : ((5,6) : (zip [] [8])))
= (1,2) : ((3,4) : ((5,6) : []))
= [(1,2), (3,4), (5,6)]
```

#### Recursión sobre varios argumentos: La función drop

- Eliminación de elementos iniciales:

```
----- Prelude -----
drop :: Int -> [a] -> [a]
drop 0 xs = xs
drop (n+1) [] = []
drop (n+1) (x:xs) = drop n xs
```

- Cálculo:

drop 2 [5,7,9,4]		drop 5 [1,4]
= drop 1 [7,9,4]		= drop 4 [4]
= drop 0 [9,4]		= drop 1 []
= [9,4]		= []

## 6.4. Recursión múltiple

### Recursión múltiple: La función de Fibonacci

- La sucesión de Fibonacci es: 0,1,1,2,3,5,8,13,21,... Sus dos primeros términos son 0 y 1 y los restantes se obtienen sumando los dos anteriores.
- (fibonacci n) es el n-ésimo término de la sucesión de Fibonacci. Por ejemplo,

```
fibonacci 8 ~> 21
```

```
fibonacci :: Int -> Int
fibonacci 0     = 0
fibonacci 1     = 1
fibonacci (n+2) = fibonacci n + fibonacci (n+1)
```

### Recursión múltiple: Ordenación rápida

- Algoritmo de ordenación rápida:

```
ordena :: (Ord a) => [a] -> [a]
ordena [] = []
ordena (x:xs) =
  (ordena menores) ++ [x] ++ (ordena mayores)
  where menores = [a | a <- xs, a <= x]
        mayores = [b | b <- xs, b > x]
```

## 6.5. Recursión mutua

### Recursión mutua: Par e impar

- Par e impar por recursión mutua:

```

par :: Int -> Bool
par 0      = True
par (n+1) = impar n

impar :: Int -> Bool
impar 0    = False
impar (n+1) = par n

```

■ Cálculo:

impar 3		par 3
= par 2		= impar 2
= impar 1		= par 1
= par 0		= impar 0
= True		= False

### Recursión mutua: Posiciones pares e impares

- (pares xs) son los elementos de xs que ocupan posiciones pares.
- (impares xs) son los elementos de xs que ocupan posiciones impares.

```

pares :: [a] -> [a]
pares []      = []
pares (x:xs) = x : impares xs

impares :: [a] -> [a]
impares []    = []
impares (_:xs) = pares xs

```

■ Cálculo:

```

pares [1,3,5,7]
= 1:(impares [3,5,7])
= 1:(pares [5,7])
= 1:(5:(impares [7]))
= 1:(5:[])
= [1,5]

```

## 6.6. Heurísticas para las definiciones recursivas

### Aplicación del método: La función `product`

- Paso 1: Definir el tipo:

```
product :: [Int] -> Int
```

- Paso 2: Enumerar los casos:

```
product :: [Int] -> Int
product []      =
product (n:ns) =
```

- Paso 3: Definir los casos simples:

```
product :: [Int] -> Int
product []      = 1
product (n:ns) =
```

- Paso 4: Definir los otros casos:

```
product :: [Int] -> Int
product []      = 1
product (n:ns) = n * product ns
```

- Paso 5: Generalizar y simplificar:

```
product :: Num a => [a] -> a
product = foldr (*) 1
```

donde (`foldr op e l`) pliega por la derecha la lista `l` colocando el operador `op` entre sus elementos y el elemento `e` al final. Por ejemplo,

```
foldr (+) 6 [2,3,5] ~> 2+(3+(5+6)) ~> 16
foldr (-) 6 [2,3,5] ~> 2-(3-(5-6)) ~> -2
```



**Aplicación del método: La función drop**

- Paso 1: Definir el tipo:

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

- Paso 2: Enumerar los casos:

```
drop :: Int -> [a] -> [a]
drop 0 []           =
drop 0 (x:xs)      =
drop (n+1) []      =
drop (n+1) (x:xs) =
```

- Paso 3: Definir los casos simples:

```
drop :: Int -> [a] -> [a]
drop 0 []           = []
drop 0 (x:xs)      = x:xs
drop (n+1) []      = []
drop (n+1) (x:xs) =
```

- Paso 4: Definir los otros casos:

```
drop :: Int -> [a] -> [a]
drop 0 []           = []
drop 0 (x:xs)      = x:xs
drop (n+1) []      = []
drop (n+1) (x:xs) = drop n xs
```

- Paso 5: Generalizar y simplificar:

```
drop :: Integral b => b -> [a] -> [a]
drop 0 xs           = xs
drop (n+1) []      = []
drop (n+1) (_:xs) = drop n xs
```

**Aplicación del método: La función init**

- init elimina el último elemento de una lista no vacía.

- Paso 1: Definir el tipo:

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

- Paso 2: Enumerar los casos:

```
init :: [a] -> [a]
init (x:xs) =
```

- Paso 3: Definir los casos simples:

```
init :: [a] -> [a]
init (x:xs) | null xs    = []
            | otherwise =
```

- Paso 4: Definir los otros casos:

```
init :: [a] -> [a]
init (x:xs) | null xs    = []
            | otherwise = x : init xs
```

- Paso 5: Generalizar y simplificar:

```
init :: [a] -> [a]
init []    = []
init (x:xs) = x : init xs
```

## Bibliografía

1. R. Bird. *Introducción a la programación funcional con Haskell*. Prentice Hall, 2000.
  - Cap. 3: Números.
  - Cap. 4: Listas.
2. G. Hutton *Programming in Haskell*. Cambridge University Press, 2007.
  - Cap. 6: Recursive functions.
3. B. O'Sullivan, D. Stewart y J. Goerzen *Real World Haskell*. O'Reilly, 2008.
  - Cap. 2: Types and Functions.

- 
4. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. *Razonando con Haskell*. Thompson, 2004.
    - Cap. 2: Introducción a Haskell.
    - Cap. 6: Programación con listas.
  5. S. Thompson. *Haskell: The Craft of Functional Programming*, Second Edition. Addison-Wesley, 1999.
    - Cap. 4: Designing and writing programs.



# Tema 7

## Razonamiento sobre programas

### Contenido

---

<b>7.1. Razonamiento ecuacional</b>	<b>63</b>
7.1.1. Cálculo con longitud	63
7.1.2. Propiedad de intercambia	63
7.1.3. Inversa de listas unitarias	64
7.1.4. Razonamiento ecuacional con análisis de casos	65
<b>7.2. Razonamiento por inducción sobre los naturales</b>	<b>65</b>
7.2.1. Esquema de inducción sobre los naturales	65
7.2.2. Ejemplo de inducción sobre los naturales	66
<b>7.3. Razonamiento por inducción sobre listas</b>	<b>67</b>
7.3.1. Esquema de inducción sobre listas	67
7.3.2. Asociatividad de ++	67
7.3.3. [] es la identidad para ++ por la derecha	68
7.3.4. Relación entre length y ++	69
7.3.5. Relación entre take y drop	70
7.3.6. La concatenación de listas vacías es vacía	71
<b>7.4. Equivalencia de funciones</b>	<b>72</b>

---

### 7.1. Razonamiento ecuacional

#### 7.1.1. Cálculo con longitud

- Programa:

```
longitud []      = 0                -- longitud.1
longitud (_:xs) = 1 + longitud xs  -- longitud.2
```

- Propiedad: `longitud [2,3,1] = 3`

- Demostración:

```
longitud [2,3,1]
= 1 + longitud [2,3]           [por longitud.2]
= 1 + (1 + longitud [3])      [por longitud.2]
= 1 + (1 + (1 + longitud [])) [por longitud.2]
= 1 + (1 + (1 + 0))           [por longitud.1]
= 3
```

### 7.1.2. Propiedad de intercambia

- Programa:

```
intercambia :: (a,b) -> (b,a)
intercambia (x,y) = (y,x)      -- intercambia
```

- Propiedad: `intercambia (intercambia (x,y)) = (x,y)`.

- Demostración:

```
intercambia (intercambia (x,y))
= intercambia (y,x)           [por intercambia]
= (x,y)                       [por intercambia]
```

#### Comprobación con QuickCheck

- Propiedad:

```
prop_intercambia :: Eq a => a -> a -> Bool
prop_intercambia x y =
  intercambia (intercambia (x,y)) == (x,y)
```

- Comprobación:

```
*Main> quickCheck prop_intercambia
+++ OK, passed 100 tests.
```

### 7.1.3. Inversa de listas unitarias

- Inversa de una lista:

```

inversa :: [a] -> [a]
inversa []      = []                -- inversa.1
inversa (x:xs) = inversa xs ++ [x] -- inversa.2

```

- Prop.:  $\text{inversa } [x] = [x]$ 

```

inversa [x]
= inversa (x:[])      [notación de lista]
= (inversa []) ++ [x] [inversa.2]
= [] ++ [x]          [inversa.1]
= [x]                [def. de ++]

```

Comprobación con QuickCheck

- Propiedad:

```

prop_inversa_unitaria :: (Eq a) => a -> Bool
prop_inversa_unitaria x =
  inversa [x] == [x]

```

- Comprobación:

```

*Main> quickCheck prop_inversa_unitaria
+++ OK, passed 100 tests.

```

### 7.1.4. Razonamiento ecuacional con análisis de casos

- Negación lógica:

```

----- Prelude -----
not :: Bool -> Bool
not False = True
not True  = False

```

- Prop.:  $\text{not } (\text{not } x) = x$

- Demostración por casos:

- Caso 1:  $x = \text{True}$ :

```

not (not True) = not False [not.2]
               = True      [not.1]

```

- Caso 2:  $x = \text{False}$ :
  - `not (not False) = not True [not.1]`
  - `= False     [not.2]`

Comprobación con QuickCheck

- Propiedad:

```
prop_doble_negacion :: Bool -> Bool
prop_doble_negacion x =
  not (not x) == x
```

- Comprobación:

```
*Main> quickCheck prop_doble_negacion
+++ OK, passed 100 tests.
```

## 7.2. Razonamiento por inducción sobre los naturales

### 7.2.1. Esquema de inducción sobre los naturales

Para demostrar que todos los números naturales tienen una propiedad  $P$  basta probar:

1. Caso base  $n=0$ :  
 $P(0)$ .
2. Caso inductivo  $n=(m+1)$ :  
Suponiendo  $P(m)$  demostrar  $P(m+1)$ .

En el caso inductivo, la propiedad  $P(n)$  se llama la hipótesis de inducción.

### 7.2.2. Ejemplo de inducción sobre los naturales

#### Ejemplo de inducción sobre los naturales: Propiedad

- `(replicate n x)` es la lista formada por  $n$  elementos iguales a  $x$ . Por ejemplo,

```
| replicate 3 5 ~> [5,5,5]
```

\_\_\_\_\_ Prelude \_\_\_\_\_

```
replicate :: Int -> a -> [a]
replicate 0 _ = []
replicate (n+1) x = x : replicate n x
```

- Prop.: `length (replicate n xs) = n`



### Ejemplo de inducción sobre los naturales: Demostración

- Caso base (n=0):
  - length (replicate 0 xs)
  - = length [] [por replicate.1]
  - = 0 [por def. length]
  
- Caso inductivo (n=m+1):
  - length (replicate (m+1) xs)
  - = length (x:(replicate m xs)) [por replicate.2]
  - = 1 + length (replicate m xs) [por def. length]
  - = 1 + m [por hip. ind.]
  - = m + 1 [por conmutativa de +]

### Ejemplo de inducción sobre los naturales: Verificación

Verificación con QuickCheck:

- Especificación de la propiedad:

```
prop_length_replicate :: Int -> Int -> Bool
prop_length_replicate n xs =
  length (replicate m xs) == m
  where m = abs n
```

- Comprobación de la propiedad:

```
*Main> quickCheck prop_length_replicate
OK, passed 100 tests.
```

## 7.3. Razonamiento por inducción sobre listas

### 7.3.1. Esquema de inducción sobre listas

Para demostrar que todas las listas finitas tienen una propiedad P basta probar:

1. Caso base  $xs=[]$ :  
P([]).
2. Caso inductivo  $xs=(y:ys)$ :  
Suponiendo P(ys) demostrar P(y:ys).

En el caso inductivo, la propiedad P(ys) se llama la hipótesis de inducción.

### 7.3.2. Asociatividad de ++

- Programa:

```

Prelude
(++ ) :: [a] -> [a] -> [a]
[]      ++ ys = ys          -- ++.1
(x:xs) ++ ys = x : (xs ++ ys) -- ++.2

```

- Propiedad:  $xs++(ys++zs)=(xs++ys)++zs$

- Comprobación con QuickCheck:

```

prop_asociativa_conc :: [Int] -> [Int] -> [Int] -> Bool
prop_asociativa_conc xs ys zs =
  xs++(ys++zs)==(xs++ys)++zs

```

```

Main> quickCheck prop_asociatividad_conc
OK, passed 100 tests.

```

- Demostración por inducción en xs:

- Caso base  $xs=[]$ : Reduciendo el lado izquierdo

$$\begin{aligned}
 & xs++(ys++zs) \\
 &= []++(ys++zs) && \text{[por hipótesis]} \\
 &= ys++zs && \text{[por ++.1]}
 \end{aligned}$$

y reduciendo el lado derecho

$$\begin{aligned}
 & (xs++ys)++zs \\
 &= ([]++ys)++zs && \text{[por hipótesis]} \\
 &= ys++zs && \text{[por ++.1]}
 \end{aligned}$$

Luego,  $xs++(ys++zs)=(xs++ys)++zs$

- Demostración por inducción en xs:

- Caso inductivo  $xs=a:as$ : Suponiendo la hipótesis de inducción

$as++(ys++zs)=(as++ys)++zs$  hay que demostrar que

$$\begin{aligned}
 & (a:as)++(ys++zs) = ((a:as)++ys)++zs \\
 & (a:as)++(ys++zs) \\
 &= a:(as++(ys++zs)) && \text{[por ++.2]} \\
 &= a:((as++ys)++zs) && \text{[por hip. ind.]} \\
 &= (a:(as++ys))++zs && \text{[por ++.2]} \\
 &= ((a:as)++ys)++zs && \text{[por ++.2]}
 \end{aligned}$$

### 7.3.3. [] es la identidad para ++ por la derecha

- Propiedad:  $xs++[]=xs$
- Comprobación con QuickCheck:

```
prop_identidad_concatenacion :: [Int] -> Bool
prop_identidad_concatenacion xs = xs++[] == xs
```

```
Main> quickCheck prop_identidad_concatenacion
OK, passed 100 tests.
```

- Demostración por inducción en  $xs$ :
  - Caso base  $xs=[]$ :
    - =  $[]++[]$
    - =  $[]$  [por ++.1]
  - Caso inductivo  $xs=(a:as)$ : Suponiendo la hipótesis de inducción
    - $as++[]=as$  hay que demostrar que
    - $(a:as)++[]=(a:as)$
    - $(a:as)++[]$
    - =  $a:(as++[])$  [por ++.2]
    - =  $a:as$  [por hip. ind.]

### 7.3.4. Relación entre length y ++

- Programas:

```
length :: [a] -> Int
length [] = 0 -- length.1
length (x:xs) = 1 + n_length xs -- length.2

(++): :: [a] -> [a] -> [a]
[] ++ ys = ys -- ++.1
(x:xs) ++ ys = x : (xs ++ ys) -- ++.2
```

- Propiedad:  $length(xs++ys)=(length\ xs)+(length\ ys)$
- Comprobación con QuickCheck:

```
prop_length_append :: [Int] -> [Int] -> Bool
prop_length_append xs ys = length(xs++ys)==(length xs)+(length ys)
```

```
Main> quickCheck prop_length_append
OK, passed 100 tests.
```

■ Demostración por inducción en `xs`:

- Caso base `xs=[]`:
 

<code>length ([]++ys)</code>	
<code>= length ys</code>	[por ++.1]
<code>= 0+(length ys)</code>	[por aritmética]
<code>= (length [])+(length ys)</code>	[por length.1]

■ Demostración por inducción en `xs`:

- Caso inductivo `xs=(a:as)`: Suponiendo la hipótesis de inducción
 

<code>length(as++ys) = (length as)+(length ys)</code>	
---	--

 hay que demostrar que
 

<code>length((a:as)++ys) = (length (a:as))+(length ys)</code>	
<code>length((a:as)++ys)</code>	
<code>= length(a:(as++ys))</code>	[por ++.2]
<code>= 1 + length(as++ys)</code>	[por length.2]
<code>= 1 + ((length as) + (length ys))</code>	[por hip. ind.]
<code>= (1 + (length as)) + (length ys)</code>	[por aritmética]
<code>= (length (a:as)) + (length ys)</code>	[por length.2]

### 7.3.5. Relación entre `take` y `drop`

■ Programas:

```
take :: Int -> [a] -> [a]
take 0 _      = []           -- take.1
take _ []     = []           -- take.2
take n (x:xs) = x : take (n-1) xs -- take.3

drop :: Int -> [a] -> [a]
drop 0 xs     = xs           -- drop.1
drop _ []     = []           -- drop.2
drop n (_:xs) = drop (n-1) xs -- drop.3

(++):: [a] -> [a] -> [a]
[] ++ ys     = ys           -- ++.1
(x:xs) ++ ys = x : (xs ++ ys) -- ++.2
```

- Propiedad: `take n xs ++ drop n xs = xs`
- Comprobación con QuickCheck:

```
prop_take_drop :: Int -> [Int] -> Property
prop_take_drop n xs =
  n >= 0 ==> take n xs ++ drop n xs == xs
```

```
Main> quickCheck prop_take_drop
OK, passed 100 tests.
```

- Demostración por inducción en  $n$ :
  - Caso base  $n=0$ :
 
$$\begin{aligned} & \text{take } 0 \text{ xs ++ drop } 0 \text{ xs} \\ &= [] ++ \text{xs} && \text{[por take.1 y drop.1]} \\ &= \text{xs} && \text{[por ++.1]} \end{aligned}$$
  - Caso inductivo  $n=m+1$ : Suponiendo la hipótesis de inducción 1
 
$$(\forall \text{xs} :: [a]) \text{take } m \text{ xs ++ drop } m \text{ xs} = \text{xs}$$
 hay que demostrar que
 
$$(\forall \text{xs} :: [a]) \text{take } (m+1) \text{ xs ++ drop } (m+1) \text{ xs} = \text{xs}$$

Lo demostraremos por inducción en  $\text{xs}$ :

- Caso base  $\text{xs}=[]$ :
 
$$\begin{aligned} & \text{take } (m+1) [] ++ \text{drop } (m+1) [] \\ &= [] ++ [] && \text{[por take.2 y drop.2]} \\ &= [] && \text{[por ++.1]} \end{aligned}$$
- Caso inductivo  $\text{xs}=(a:\text{as})$ : Suponiendo la hip. de inducción 2
 
$$\text{take } (m+1) \text{ as ++ drop } (m+1) \text{ as} = \text{as}$$
 hay que demostrar que
 
$$\begin{aligned} & \text{take } (m+1) (a:\text{as}) ++ \text{drop } (m+1) (a:\text{as}) = (a:\text{as}) \\ & \text{take } (m+1) (a:\text{as}) ++ \text{drop } (m+1) (a:\text{as}) \\ &= (a:(\text{take } m \text{ as})) ++ (\text{drop } m \text{ as}) && \text{[take.3 y drop.3]} \\ &= (a:(\text{take } m \text{ as} ++ (\text{drop } m \text{ as}))) && \text{[por ++.2]} \\ &= a:\text{as} && \text{[por hip. de ind. 1]} \end{aligned}$$

### 7.3.6. La concatenación de listas vacías es vacía

- Programas:

```

----- Prelude -----
null :: [a] -> Bool
null []          = True           -- null.1
null (_:_)      = False          -- null.2

(++) :: [a] -> [a] -> [a]
[] ++ ys        = ys             -- (++).1
(x:xs) ++ ys    = x : (xs ++ ys) -- (++).2

```

- Propiedad: `null xs = null (xs ++ xs)`.
- Demostración por inducción en `xs`:

- Caso 1: `xs = []`: Reduciendo el lado izquierdo

```

null xs
= null []      [por hipótesis]
= True         [por null.1]

```

y reduciendo el lado derecho

```

null (xs ++ xs)
= null ([] ++ []) [por hipótesis]
= null []         [por (++).1]
= True           [por null.1]

```

Luego, `null xs = null (xs ++ xs)`.

- Demostración por inducción en `xs`:

- Caso `xs = (y:ys)`: Reduciendo el lado izquierdo

```

null xs
= null (y:ys) [por hipótesis]
= False      [por null.2]

```

y reduciendo el lado derecho

```

null (xs ++ xs)
= null ((y:ys) ++ (y:ys)) [por hipótesis]
= null (y:(ys ++ (y:ys))) [por (++).2]
= False                  [por null.2]

```

Luego, `null xs = null (xs ++ xs)`.

## 7.4. Equivalencia de funciones

- Programas:

```

inversa1, inversa2 :: [a] -> [a]
inversa1 []      = []                -- inversa1.1
inversa1 (x:xs) = inversa1 xs ++ [x] -- inversa1.2

inversa2 xs = inversa2Aux xs []      -- inversa2.1
  where inversa2Aux []      ys = ys   -- inversa2Aux.1
        inversa2Aux (x:xs) ys = inversa2Aux xs (x:ys) -- inversa2Aux.2

```

- Propiedad:  $\text{inversa1 } xs = \text{inversa2 } xs$

- Comprobación con QuickCheck:

```

prop_equiv_inversa :: [Int] -> Bool
prop_equiv_inversa xs = inversa1 xs == inversa2 xs

```

- Demostración: Es consecuencia del siguiente lema:

$$\text{inversa1 } xs ++ ys = \text{inversa2Aux } xs \text{ } ys$$

En efecto,

```

inversa1 xs
= inversa1 xs ++ []      [por identidad de ++]
= inversa2Aux xs ++ []  [por el lema]
= inversa2 xs           [por el inversa2.1]

```

- Demostración del lema: Por inducción en  $xs$ :

- Caso base  $xs=[]$ :

```

inversa1 [] ++ ys
= [] ++ ys          [por inversa1.1]
= ys                [por ++.1]
= inversa2Aux [] ++ ys [por inversa2Aux.1]

```

- Caso inductivo  $xs=(a:as)$ : La hipótesis de inducción es

$$(\forall ys :: [a]) \text{inversa1 } as ++ ys = \text{inversa2Aux } as \text{ } ys$$

Por tanto,

```
inversa1 (a:as) ++ ys
= (inversa1 as ++ [a]) ++ ys      [por inversa1.2]
= (inversa1 as) ++ ([a] ++ ys)   [por asociativa de ++]
= (inversa1 as) ++ (a:ys)        [por ley unitaria]
= (inversa2Aux as (a:ys))         [por hip. de inducción]
= (inversa2Aux (a:as) ys)         [por inversa2Aux.2]
```

## Bibliografía

1. H. C. Cunningham (2007) *Notes on Functional Programming with Haskell*.
2. J. Fokker (1996) *Programación funcional*.
3. G. Hutton *Programming in Haskell*. Cambridge University Press, 2007.
  - Cap. 13: Reasoning about programs.
4. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. *Razonando con Haskell*. Thompson, 2004.
  - Cap. 6: Programación con listas.
5. S. Thompson. *Haskell: The Craft of Functional Programming*, Second Edition. Addison-Wesley, 1999.
  - Cap. 8: Reasoning about programs.
6. E.P. Wentworth (1994) *Introduction to Funcional Programming*.



# Tema 8

## Funciones de orden superior

### Contenido

---

<b>8.1. Funciones de orden superior</b> . . . . .	75
<b>8.2. Procesamiento de listas</b> . . . . .	76
8.2.1. La función <code>map</code> . . . . .	76
8.2.2. La función <code>filter</code> . . . . .	78
<b>8.3. Función de plegado por la derecha: <code>foldr</code></b> . . . . .	79
<b>8.4. Función de plegado por la izquierda: <code>foldl</code></b> . . . . .	82
<b>8.5. Composición de funciones</b> . . . . .	83
<b>8.6. Caso de estudio: Codificación binaria y transmisión de cadenas</b> . . . . .	84
8.6.1. Cambio de bases . . . . .	84
8.6.2. Codificación y decodificación . . . . .	86

---

### 8.1. Funciones de orden superior

#### Funciones de orden superior

- Una función es **de orden superior** si toma una función como argumento o devuelve una función como resultado.
- `(dosVeces f x)` es el resultado de aplicar `f` a `f x`. Por ejemplo,

<code>dosVeces (*3) 2</code>	$\rightsquigarrow$	<code>18</code>
<code>dosVeces reverse [2,5,7]</code>	$\rightsquigarrow$	<code>[2,5,7]</code>

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

- Prop: `dosVeces reverse = id`  
donde `id` es la función identidad.

\_\_\_\_\_ Prelude \_\_\_\_\_

```
id :: a -> a
id x = x
```

## Usos de las funciones de orden superior

- Definición de patrones de programación.
  - Aplicación de una función a todos los elementos de una lista.
  - Filtrado de listas por propiedades.
  - Patrones de recursión sobre listas.
- Diseño de lenguajes de dominio específico:
  - Lenguajes para procesamiento de mensajes.
  - Analizadores sintácticos.
  - Procedimientos de entrada/salida.
- Uso de las propiedades algebraicas de las funciones de orden superior para razonar sobre programas.

## 8.2. Procesamiento de listas

### 8.2.1. La función `map`

#### La función `map`: Definición

- `(map f xs)` es la lista obtenida aplicando `f` a cada elemento de `xs`. Por ejemplo,

```
map (*2) [3,4,7]    ~> [6,8,14]
map sqrt [1,2,4]   ~> [1.0,1.4142135623731,2.0]
map even [1..5]    ~> [False,True,False,True,False]
```

- Definición de `map` por comprensión:

```
map :: (a -> b) -> [a] -> [b]
map f xs = [f x | x <- xs]
```

- Definición de map por recursión:

```
----- Prelude -----
map :: (a -> b) -> [a] -> [b]
map _ []      = []
map f (x:xs) = f x : map f xs
```

### Relación entre sum y map

- La función sum:

```
----- Prelude -----
sum :: [Int] -> Int
sum []      = 0
sum (x:xs) = x + sum xs
```

- Propiedad:  $\text{sum } (\text{map } (2*) \text{ xs}) = 2 * \text{sum xs}$
- Comprobación con QuickCheck:

```
prop_sum_map :: [Int] -> Bool
prop_sum_map xs = sum (map (2*) xs) == 2 * sum xs
```

```
*Main> quickCheck prop_sum_map
+++ OK, passed 100 tests.
```

### Demostración de la propiedad por inducción en xs

- Caso []:
 

sum (map (2*) xs)	
= sum (map (2*) [])	[por hipótesis]
= sum []	[por map.1]
= 0	[por sum.1]
= 2 * 0	[por aritmética]
= 2 * sum []	[por sum.1]
= 2 * sum xs	[por hipótesis]

- Caso  $xs = (y : ys)$ : Entonces,
 

<code>sum (map (2*) xs)</code>	
<code>= sum (map (2*) (y : ys))</code>	[por hipótesis]
<code>= sum (2*) y : (map (2*) ys)</code>	[por map. 2]
<code>= (2*) y + (sum (map (2*) ys))</code>	[por sum. 2]
<code>= (2*) y + (2 * sum ys)</code>	[por hip. de inducción]
<code>= (2 * y) + (2 * sum ys)</code>	[por (2*)]
<code>= 2 * (y + sum ys)</code>	[por aritmética]
<code>= 2 * sum (y : ys)</code>	[por sum. 2]
<code>= 2 * sum xs</code>	[por hipótesis]

### Comprobación de propiedades con argumentos funcionales

- La aplicación de una función a los elementos de una lista conserva su longitud:

```
prop_map_length (Function _ f) xs =
  length (map f xs) == length xs
```

- En el inicio del fichero hay que escribir

```
import Test.QuickCheck.Function
```

- Comprobación

```
*Main> quickCheck prop_map_length
+++ OK, passed 100 tests.
```

### 8.2.2. La función filter

#### La función filter

- `filter p xs` es la lista de los elementos de `xs` que cumplen la propiedad `p`. Por ejemplo,

```
filter even [1,3,5,4,2,6,1] ~> [4,2,6]
filter (>3) [1,3,5,4,2,6,1] ~> [5,4,6]
```

- Definición de `filter` por comprensión:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [x | x <- xs, p x]
```

- Definición de `filter` por recursión:

```

Prelude
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs) | p x = x : filter p xs
                | otherwise = filter p xs

```

### Uso conjunto de `map` y `filter`

- `sumaCuadradosPares xs` es la suma de los cuadrados de los números pares de la lista `xs`. Por ejemplo,

```

sumaCuadradosPares [1..5] ~> 20

```

```

sumaCuadradosPares :: [Int] -> Int
sumaCuadradosPares xs = sum (map (^2) (filter even xs))

```

- Definición por comprensión:

```

sumaCuadradosPares' :: [Int] -> Int
sumaCuadradosPares' xs = sum [x^2 | x <- xs, even x]

```

### Predefinidas de orden superior para procesar listas

- `all p xs` se verifica si todos los elementos de `xs` cumplen la propiedad `p`. Por ejemplo,

```

all odd [1,3,5] ~> True
all odd [1,3,6] ~> False

```

- `any p xs` se verifica si algún elemento de `xs` cumple la propiedad `p`. Por ejemplo,

```

any odd [1,3,5] ~> True
any odd [2,4,6] ~> False

```

- `takeWhile p xs` es la lista de los elementos iniciales de `xs` que verifican el predicado `p`. Por ejemplo,

```

takeWhile even [2,4,6,7,8,9] ~> [2,4,6]

```

- `dropWhile p xs` es la lista `xs` sin los elementos iniciales que verifican el predicado `p`. Por ejemplo,

```

dropWhile even [2,4,6,7,8,9] ~> [7,8,9]

```

### 8.3. Función de plegado por la derecha: `foldr`

#### Esquema básico de recursión sobre listas

- Ejemplos de definiciones recursivas:

```

----- Prelude -----
sum []           = 0
sum (x:xs)      = x + sum xs
product []      = 1
product (x:xs) = x * product xs
or []           = False
or (x:xs)      = x || or xs
and []          = True
and (x:xs)     = x && and xs

```

- Esquema básico de recursión sobre listas:

```

f []           = v
f (x:xs) = x 'op' (f xs)

```

#### El patrón `foldr`

- Redefiniciones con el patrón `foldr`

```

sum      = foldr (+) 0
product = foldr (*) 1
or       = foldr (||) False
and      = foldr (&&) True

```

- Definición del patrón `foldr`

```

----- Prelude -----
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f v []      = v
foldr f v (x:xs) = f x (foldr f v xs)

```

#### Visión no recursiva de `foldr`

- Cálculo con sum:
 

```
sum [2,3,5]
= foldr (+) 0 [2,3,5]           [def. de sum]
= foldr (+) 0 2:(3:(5:[]))     [notación de lista]
=                               2+(3+(5+0)) [sustituir (:) por (+) y [] por 0]
= 10                           [aritmética]
```
- Cálculo con sum:
 

```
product [2,3,5]
= foldr (*) 1 [2,3,5]          [def. de sum]
= foldr (*) 1 2:(3:(5:[]))    [notación de lista]
=                               2*(3*(5*1)) [sustituir (:) por (*) y [] por 1]
= 30                           [aritmética]
```
- Cálculo de foldr f v xs  
 Sustituir en xs los (:) por f y [] por v.

### Definición de la longitud mediante foldr

- Ejemplo de cálculo de la longitud:

```
longitud [2,3,5]
= longitud 2:(3:(5:[]))
=           1+(1+(1+0))           [Sustituciones]
= 3
```

- Sustituciones:

- los (:) por (\ \_ n -> 1+n)
- la [] por 0

- Definición de length usando foldr

```
longitud :: [a] -> Int
longitud = foldr (\ _ n -> 1+n) 0
```

### Definición de la inversa mediante foldr

- Ejemplo de cálculo de la inversa:

```
inversa [2,3,5]
= inversa 2:(3:(5:[]))
=           (([] ++ [5]) ++ [3]) ++ [2] [Sustituciones]
= [5,3,2]
```

- Sustituciones:
  - los `(:)` por `(\ x xs -> xs ++ [x])`
  - la `[]` por `[]`
- Definición de inversa usando `foldr`

```

inversa :: [a] -> [a]
inversa = foldr (\ x xs -> xs ++ [x]) []

conc xs ys = (foldr (: ) ys) xs

```

### Definición de la concatenación mediante `foldr`

- Ejemplo de cálculo de la concatenación:

```

conc [2,3,5] [7,9]
= conc 2:(3:(5:[])) [7,9]
=      2:(3:(5:[7,9]))      [Sustituciones]
= [2,3,5,7,9]

```

- Sustituciones:
  - los `(:)` por `(:)`
  - la `[]` por `ys`
- Definición de `conc` usando `foldr`

```

conc xs ys = (foldr (: ) ys) xs

```

## 8.4. Función de plegado por la izquierda: `foldl`

### Definición de suma de lista con acumuladores

- Definición de suma con acumuladores:

```

suma :: [Integer] -> Integer
suma = sumaAux 0
  where sumaAux v []      = v
        sumaAux v (x:xs) = sumaAux (v+x) xs

```

- Cálculo con suma:



```

suma [2,3,7] = sumaAux 0 [2,3,7]
              = sumaAux (0+2) [3,7]
              = sumaAux 2 [3,7]
              = sumaAux (2+3) [7]
              = sumaAux 5 [7]
              = sumaAux (5+7) []
              = sumaAux 12 []
              = 12

```

### Patrón de definición de recursión con acumulador

- Patrón de definición (generalización de sumaAux):

```

f v []      = v
f v (x:xs) = f (v*x) xs

```

- Definición con el patrón foldl:

```

suma      = foldl (+) 0
product  = foldl (*) 1
or        = foldl (||) False
and       = foldl (&&) True

```

### Definición de foldl

- Definición de foldl:

```

----- Prelude -----
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f v []      = v
foldl f v (x:xs) = foldl f (f v x) xs

```

## 8.5. Composición de funciones

### Composición de funciones

- Definición de la composición de dos funciones:

```

----- Prelude -----
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)

```

- Uso de composición para simplificar definiciones:

- Definiciones sin composición:

```
par n                = not (impar n)
doVeces f x         = f (f x )
sumaCuadradosPares ns = sum (map (^2) (filter even ns))
```

- Definiciones con composición:

```
par                = not . impar
dosVeces f        = f . f
sumaCuadradosPares = sum . map (^2) . filter even
```

### Composición de una lista de funciones

- La función identidad:

```
----- Prelude -----
id :: a -> a
id = \x -> x
```

- (composicionLista fs) es la composición de la lista de funciones fs. Por ejemplo,

```
composicionLista [(*)^(2),(^2)] 3    ~> 18
composicionLista [(^2),(*)^(2)] 3    ~> 36
composicionLista [(/9),(^2),(*)^(2)] 3 ~> 4.0
```

```
composicionLista :: [a -> a] -> (a -> a)
composicionLista = foldr (.) id
```

## 8.6. Caso de estudio: Codificación binaria y transmisión de cadenas

- Objetivos:

1. Definir una función que convierta una cadena en una lista de ceros y unos junto con otra función que realice la conversión opuesta.
2. Simular la transmisión de cadenas mediante ceros y unos.

- Los números binarios se representan mediante listas de bits en orden inverso. Un bit es cero o uno. Por ejemplo, el número 1101 se representa por [1,0,1,1].

- El tipo `Bit` es el de los bits.

```
type Bit = Int
```

### 8.6.1. Cambio de bases

#### Cambio de bases: De binario a decimal

- `(bin2int x)` es el número decimal correspondiente al número binario `x`. Por ejemplo,

```
bin2int [1,0,1,1] ~> 13
```

```
bin2int :: [Bit] -> Int
bin2int = foldr (\x y -> x + 2*y) 0
```

Por ejemplo,

```
bin2int [1,0,1,1]
= bin2int 1:(0:(1:(1:[])))
=      1+2*(0+2*(1+2*(1+2*0)))
= 13
```

#### Cambio de base: De decimal a binario

- `(int2bin x)` es el número binario correspondiente al número decimal `x`. Por ejemplo,

```
int2bin 13 ~> [1,0,1,1]
```

```
int2bin :: Int -> [Bit]
int2bin 0 = []
int2bin n = n `mod` 2 : int2bin (n `div` 2)
```

Por ejemplo,

```
int2bin 13
= 13 `mod` 2 : int2bin (13 `div` 2)
= 1 : int2bin (6 `div` 2)
= 1 : (6 `mod` 2 : int2bin (6 `div` 2))
= 1 : (0 : int2bin 3)
```

```

= 1 : (0 : (3 'mod' 2 : int2bin (3 'div' 2)))
= 1 : (0 : (1 : int2bin 1))
= 1 : (0 : (1 : (1 : int2bin 0)))
= 1 : (0 : (1 : (1 : [])))
= [1,0,1,1]

```

### Cambio de base: Comprobación de propiedades

- Propiedad: Al pasar un número natural a binario con `int2bin` y el resultado a decimal con `bin2int` se obtiene el número inicial.

```

prop_int_bin :: Int -> Bool
prop_int_bin x =
  bin2int (int2bin y) == y
  where y = abs x

```

- Comprobación:

```

*Main> quickCheck prop_int_bin
+++ OK, passed 100 tests.

```

## 8.6.2. Codificación y decodificación

### Creación de octetos

- Un octeto es un grupo de ocho bits.
- (`creaOcteto bs`) es el octeto correspondiente a la lista de bits `bs`; es decir, los 8 primeros elementos de `bs` si su longitud es mayor o igual que 8 y la lista de 8 elementos añadiendo ceros al final de `bs` en caso contrario. Por ejemplo,

```

Main*> creaOcteto [1,0,1,1,0,0,1,1,1,0,0,0]
[1,0,1,1,0,0,1,1]
Main*> creaOcteto [1,0,1,1]
[1,0,1,1,0,0,0,0]

```

```

creaOcteto :: [Bit] -> [Bit]
creaOcteto bs = take 8 (bs ++ repeat 0)

```

donde (`repeat x`) es una lista infinita cuyo único elemento es `x`.

## Codificación

- (`codifica c`) es la codificación de la cadena `c` como una lista de bits obtenida convirtiendo cada carácter en un número Unicode, convirtiendo cada uno de dichos números en un octeto y concatenando los octetos para obtener una lista de bits. Por ejemplo,

```
*Main> codifica "abc"
[1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0,1,1,0,0,0,1,1,0]
```

```
codifica :: String -> [Bit]
codifica = concat . map (creaOcteto . int2bin . ord)
```

donde (`concat xss`) es la lista obtenida concatenando la lista de listas `xss`.

## Codificación

- Ejemplo de codificación,

```
codifica "abc"
= concat . map (creaOcteto . int2bin . ord) "abc"
= concat . map (creaOcteto . int2bin . ord) ['a','b','c']
= concat [creaOcteto . int2bin . ord 'a',
          creaOcteto . int2bin . ord 'b',
          creaOcteto . int2bin . ord 'c']
= concat [creaOcteto [1,0,0,0,0,1,1],
          creaOcteto [0,1,0,0,0,1,1],
          creaOcteto [1,1,0,0,0,1,1]]
= concat [[1,0,0,0,0,1,1,0],
          [0,1,0,0,0,1,1,0],
          [1,1,0,0,0,1,1,0]]
= [1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0,1,1,0,0,0,1,1,0]
```

## Separación de octetos

- (`separaOctetos bs`) es la lista obtenida separando la lista de bits `bs` en listas de 8 elementos. Por ejemplo,

```
*Main> separaOctetos [1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0]
[[1,0,0,0,0,1,1,0],[0,1,0,0,0,1,1,0]]
```

```

separaOctetos :: [Bit] -> [[Bit]]
separaOctetos [] = []
separaOctetos bs =
    take 8 bs : separaOctetos (drop 8 bs)

```

## Descodificación

- (descodifica bs) es la cadena correspondiente a la lista de bits bs. Por ejemplo,

```

*Main> descodifica [1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0,1,1,0,0,0,1,1,0]
"abc"

```

```

descodifica :: [Bit] -> String
descodifica = map (chr . bin2int) . separaOctetos

```

Por ejemplo,

```

descodifica [1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0,1,1,0,0,0,1,1,0]
= (map (chr . bin2int) . separaOctetos)
  [1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0,1,1,0,0,0,1,1,0]
= map (chr . bin2int) [[1,0,0,0,0,1,1,0],[0,1,0,0,0,1,1,0],[1,1,0,0,0,1,1,0]]
= [(chr . bin2int) [1,0,0,0,0,1,1,0],
   (chr . bin2int) [0,1,0,0,0,1,1,0],
   (chr . bin2int) [1,1,0,0,0,1,1,0]]
= [chr 97, chr 98, chr 99]
= "abc"

```

## Transmisión

- Los canales de transmisión pueden representarse mediante funciones que transforman cadenas de bits en cadenas de bits.
- (transmite c t) es la cadena obtenida transmitiendo la cadena t a través del canal c. Por ejemplo,

```

*Main> transmite id "Texto por canal correcto"
"Texto por canal correcto"

```

```

transmite :: ([Bit] -> [Bit]) -> String -> String
transmite canal = descodifica . canal . codifica

```

### Corrección de la transmisión

- Propiedad: Al transmitir cualquier cadena por el canal identidad se obtiene la cadena.

```
prop_transmite :: String -> Bool
prop_transmite cs =
    transmite id cs == cs
```

- Comprobación de la corrección:

```
*Main> quickCheck prop_transmite
+++ OK, passed 100 tests.
```

### Bibliografía

1. R. Bird. *Introducción a la programación funcional con Haskell*. Prentice Hall, 2000.
  - Cap. 4: Listas.
2. G. Hutton *Programming in Haskell*. Cambridge University Press, 2007.
  - Cap. 7: Higher-order functions.
3. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. *Razonando con Haskell*. Thomson, 2004.
  - Cap. 8: Funciones de orden superior y polimorfismo.
4. S. Thompson. *Haskell: The Craft of Functional Programming*, Second Edition. Addison-Wesley, 1999.
  - Cap. 9: Generalization: patterns of computation.
  - Cap. 10: Functions as values.





# Tema 9

## Declaraciones de tipos y clases

### Contenido

---

9.1. Declaraciones de tipos . . . . .	91
9.2. Definiciones de tipos de datos . . . . .	93
9.3. Definición de tipos recursivos . . . . .	95
9.4. Sistema de decisión de tautologías . . . . .	99
9.5. Máquina abstracta de cálculo aritmético . . . . .	102
9.6. Declaraciones de clases y de instancias . . . . .	104

---

### 9.1. Declaraciones de tipos

#### Declaraciones de tipos como sinónimos

- Se puede definir un nuevo nombre para un tipo existente mediante una **declaración de tipo**.
- Ejemplo: Las cadenas son listas de caracteres.

```
----- Prelude -----  
type String = [Char]
```

- El nombre del tipo tiene que empezar por mayúscula.

#### Declaraciones de tipos nuevos

- Las declaraciones de tipos pueden usarse para facilitar la lectura de tipos. Por ejemplo,

- Las posiciones son pares de enteros.

```
type Pos = (Int, Int)
```

- origen es la posición (0,0).

```
origen :: Pos
origen = (0,0)
```

- (izquierda p) es la posición a la izquierda de la posición p. Por ejemplo,

```
izquierda (3,5) ~> (2,5)
```

```
izquierda :: Pos -> Pos
izquierda (x,y) = (x-1,y)
```

### Declaraciones de tipos parametrizadas

- Las declaraciones de tipos pueden tener parámetros. Por ejemplo,

- Par a es el tipo de pares de elementos de tipo a

```
type Par a = (a,a)
```

- (multiplica p) es el producto del par de enteros p. Por ejemplo,

```
multiplica (2,5) ~> 10
```

```
multiplica :: Par Int -> Int
multiplica (x,y) = x*y
```

- (copia x) es el par formado con dos copias de x. Por ejemplo,

```
copia 5 ~> (5,5)
```

```
copia :: a -> Par a
copia x = (x,x)
```

### Declaraciones anidadas de tipos

- Las declaraciones de tipos pueden anidarse. Por ejemplo,

- Las posiciones son pares de enteros.

```
type Pos = (Int, Int)
```

- Los movimientos son funciones que va de una posición a otra.

```
type Movimiento = Pos -> Pos
```

- Las declaraciones de tipo no pueden ser recursivas. Por ejemplo, el siguiente código es erróneo.

```
type Arbol = (Int, [Arbol])
```

Al intentar cargarlo da el mensaje de error

```
|Cycle in type synonym declarations
```

## 9.2. Definiciones de tipos de datos

### Definición de tipos con data

- En Haskell pueden definirse nuevos tipos mediante data.
- El tipo de los booleanos está formado por dos valores para representar lo falso y lo verdadero.

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

- El símbolo | se lee como “o”.
- Los valores False y True se llaman los **constructores** del tipo Bool.
- Los nombres de los constructores tienen que empezar por mayúscula.

### Uso de los valores de los tipos definidos

- Los valores de los tipos definidos pueden usarse como los de los predefinidos.
- Definición del tipo de movimientos:

```
data Mov = Izquierda | Derecha | Arriba | Abajo
```

- Uso como argumento: (movimiento m p) es la posición obtenida aplicando el movimiento m a la posición p. Por ejemplo,

```
|movimiento Arriba (2,5) ~> (2,6)
```

```

movimiento :: Mov -> Pos -> Pos
movimiento Izquierda (x,y) = (x-1,y)
movimiento Derecha   (x,y) = (x+1,y)
movimiento Arriba    (x,y) = (x,y+1)
movimiento Abajo     (x,y) = (x,y-1)

```

- Uso en listas: (movimientos ms p) es la posición obtenida aplicando la lista de movimientos ms a la posición p. Por ejemplo,

```

|movimientos [Arriba, Izquierda] (2,5) ~> (1,6)

```

```

movimientos :: [Mov] -> Pos -> Pos
movimientos []      p = p
movimientos (m:ms) p = movimientos ms (movimiento m p)

```

- Uso como valor: (opuesto m) es el movimiento opuesto de m.

```

|movimiento (opuesto Arriba) (2,5) ~> (2,4)

```

```

opuesto :: Mov -> Mov
opuesto Izquierda = Derecha
opuesto Derecha   = Izquierda
opuesto Arriba    = Abajo
opuesto Abajo     = Arriba

```

### Definición de tipo con constructores con parámetros

- Los constructores en las definiciones de tipos pueden tener parámetros.
- Ejemplo de definición

```

data Figura = Circulo Float | Rect Float Float

```

- Tipos de los constructores:

```

*Main> :type Circulo
Circulo :: Float -> Figura
*Main> :type Rect
Rect :: Float -> Float -> Figura

```

- Uso del tipo como valor: (`cuadrado n`) es el cuadrado de lado `n`.

```
cuadrado :: Float -> Figura
cuadrado n = Rect n n
```

- Uso del tipo como argumento: (`area f`) es el área de la figura `f`. Por ejemplo,

```
area (Circulo 1)  ~> 3.1415927
area (Circulo 2)  ~> 12.566371
area (Rect 2 5)   ~> 10.0
area (cuadrado 3) ~> 9.0
```

```
area :: Figura -> Float
area (Circulo r) = pi*r^2
area (Rect x y)  = x*y
```

### Definición de tipos con parámetros

- Los tipos definidos pueden tener parámetros.
- Ejemplo de tipo con parámetro

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

- (`divisionSegura m n`) es la división de `m` entre `n` si `n` no es cero y nada en caso contrario. Por ejemplo,

```
divisionSegura 6 3 ~> Just 2
divisionSegura 6 0 ~> Nothing
```

```
divisionSegura :: Int -> Int -> Maybe Int
divisionSegura _ 0 = Nothing
divisionSegura m n = Just (m `div` n)
```

- (`headSegura xs`) es la cabeza de `xs` si `xs` es no vacía y nada en caso contrario. Por ejemplo,

```
headSegura [2,3,5] ~> Just 2
headSegura []      ~> Nothing
```

```
headSegura :: [a] -> Maybe a
headSegura [] = Nothing
headSegura xs = Just (head xs)
```

### 9.3. Definición de tipos recursivos

#### Definición de tipos recursivos: Los naturales

- Los tipos definidos con data pueden ser recursivos.
- Los naturales se construyen con el cero y la función sucesor.

```
data Nat = Cero | Suc Nat
        deriving Show
```

- Tipos de los constructores:

```
*Main> :type Cero
Cero :: Nat
*Main> :type Suc
Suc :: Nat -> Nat
```

- Ejemplos de naturales:

```
Cero
Suc Cero
Suc (Suc Cero)
Suc (Suc (Suc Cero))
```

#### Definiciones con tipos recursivos

- (`nat2int n`) es el número entero correspondiente al número natural `n`. Por ejemplo,

```
nat2int (Suc (Suc (Suc Cero))) ~> 3
```

```
nat2int :: Nat -> Int
nat2int Cero = 0
nat2int (Suc n) = 1 + nat2int n
```

- (`int2nat n`) es el número natural correspondiente al número entero `n`. Por ejemplo,

```
int2nat 3 ~> Suc (Suc (Suc Cero))
```

```
int2nat :: Int -> Nat
int2nat 0      = Cero
int2nat (n+1) = Suc (int2nat n)
```

- (suma m n) es la suma de los número naturales m y n. Por ejemplo,

```
*Main> suma (Suc (Suc Cero)) (Suc Cero)
Suc (Suc (Suc Cero))
```

```
suma :: Nat -> Nat -> Nat
suma Cero    n = n
suma (Suc m) n = Suc (suma m n)
```

- Ejemplo de cálculo:

```
suma (Suc (Suc Cero)) (Suc Cero)
= Suc (suma (Suc Cero) (Suc Cero))
= Suc (Suc (suma Cero (Suc Cero)))
= Suc (Suc (Suc Cero))
```

### Tipo recursivo con parámetro: Las listas

- Definición del tipo lista:

```
data Lista a = Nil | Cons a (Lista a)
```

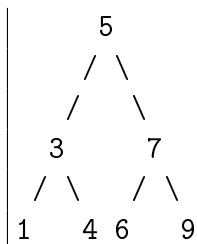
- (longitud xs) es la longitud de la lista xs. Por ejemplo,

```
longitud (Cons 2 (Cons 3 (Cons 5 Nil))) ~> 3
```

```
longitud :: Lista a -> Int
longitud Nil          = 0
longitud (Cons _ xs) = 1 + longitud xs
```

### Definición de tipos recursivos: Los árboles binarios

- Ejemplo de árbol binario:



- Definición del tipo de árboles binarios:

```
data Arbol = Hoja Int | Nodo Arbol Int Arbol
```

- Representación del ejemplo

```
ejArbol = Nodo (Nodo (Hoja 1) 3 (Hoja 4))
           5
           (Nodo (Hoja 6) 7 (Hoja 9))
```

### Definiciones sobre árboles binarios

- (ocurre m a) se verifica si m ocurre en el árbol a. Por ejemplo,

```
ocurre 4 ejArbol ~> True
ocurre 10 ejArbol ~> False
```

```
ocurre :: Int -> Arbol -> Bool
ocurre m (Hoja n)      = m == n
ocurre m (Nodo i n d) = m == n || ocurre m i || ocurre m d
```

- (aplana a) es la lista obtenida aplanando el árbol a. Por ejemplo,

```
aplana ejArbol ~> [1,3,4,5,6,7,9]
```

```
aplana :: Arbol -> [Int]
aplana (Hoja n)      = [n]
aplana (Nodo i n d) = aplana i ++ [n] ++ aplana d
```



### Definiciones sobre árboles binarios

- Un árbol es ordenado si el valor de cada nodo es mayor que los de su subárbol izquierdo y mayor que los de su subárbol derecho.
- El árbol del ejemplo es ordenado.
- `(ocurreEnArbolOrdenado m a)` se verifica si `m` ocurre en el árbol ordenado `a`. Por ejemplo,

```
ocurreEnArbolOrdenado 4 ejArbol ~> True
ocurreEnArbolOrdenado 10 ejArbol ~> False
```

```
ocurreEnArbolOrdenado :: Int -> Arbol -> Bool
ocurreEnArbolOrdenado m (Hoja n) = m == n
ocurreEnArbolOrdenado m (Nodo i n d)
  | m == n      = True
  | m < n      = ocurreEnArbolOrdenado m i
  | otherwise  = ocurreEnArbolOrdenado m d
```

### Definiciones de distintos tipos de árboles

- Árboles binarios con valores en las hojas:

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

- Árboles binarios con valores en los nodos:

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

- Árboles binarios con valores en las hojas y en los nodos:

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

- Árboles con un número variable de sucesores:

```
data Arbol a = Nodo a [Arbol a]
```

## 9.4. Sistema de decisión de tautologías

### Sintaxis de la lógica proposicional

- Definición de fórmula proposicional:
  - Las variables proposicionales son fórmulas.
  - Si  $F$  es una fórmula, entonces  $\neg F$  también lo es.
  - Si  $F$  y  $G$  son fórmulas, entonces  $F \wedge G$  y  $F \rightarrow G$  también lo son.
- Tipo de dato de fórmulas proposicionales:

```
data FProp = Const Bool
           | Var Char
           | Neg FProp
           | Conj FProp FProp
           | Impl FProp FProp
           deriving Show
```

- Ejemplos de fórmulas proposicionales:

1.  $A \wedge \neg A$
2.  $(A \wedge B) \rightarrow A$
3.  $A \rightarrow (A \wedge B)$
4.  $(A \rightarrow (A \rightarrow B)) \rightarrow B$

```
p1, p2, p3, p4 :: FProp
p1 = Conj (Var 'A') (Neg (Var 'A'))
p2 = Impl (Conj (Var 'A') (Var 'B')) (Var 'A')
p3 = Impl (Var 'A') (Conj (Var 'A') (Var 'B'))
p4 = Impl (Conj (Var 'A') (Impl (Var 'A') (Var 'B'))) (Var 'B')
```

### Semántica de la lógica proposicional

- Tablas de verdad de las conectivas:

$i$	$\neg i$	$i$	$j$	$i \wedge j$	$i \rightarrow j$
T	F	T	T	T	T
F	T	T	F	F	F
		F	T	F	T
		F	F	F	T

- Tabla de verdad para  $(A \rightarrow B) \vee (B \rightarrow A)$ :

$A$	$B$	$(A \rightarrow B)$	$(B \rightarrow A)$	$(A \rightarrow B) \vee (B \rightarrow A)$
$T$	$T$	$T$	$T$	$T$
$T$	$F$	$F$	$T$	$T$
$F$	$T$	$T$	$F$	$T$
$F$	$F$	$T$	$T$	$T$

- Las interpretaciones son listas formadas por el nombre de una variable proposicional y un valor de verdad.

```
type Interpretacion = [(Char, Bool)]
```

- `(valor i p)` es el valor de la fórmula  $p$  en la interpretación  $i$ . Por ejemplo,

```
valor [('A',False),('B',True)] p3 ~> True
valor [('A',True),('B',False)] p3 ~> False
```

```
valor :: Interpretacion -> FProp -> Bool
valor _ (Const b) = b
valor i (Var x)   = busca x i
valor i (Neg p)   = not (valor i p)
valor i (Conj p q) = valor i p && valor i q
valor i (Impl p q) = valor i p <= valor i q
```

- `(busca c t)` es el valor del primer elemento de la lista de asociación  $t$  cuya clave es  $c$ . Por ejemplo,

```
busca 2 [(1,'a'),(3,'d'),(2,'c')] ~> 'c'
```

```
busca :: Eq c => c -> [(c,v)] -> v
busca c t = head [v | (c',v) <- t, c == c']
```

- `(variables p)` es la lista de los nombres de las variables de  $p$ .

```
variables p3 ~> "AAB"
```

```
variables :: FProp -> [Char]
variables (Const _) = []
variables (Var x)   = [x]
variables (Neg p)   = variables p
variables (Conj p q) = variables p ++ variables q
variables (Impl p q) = variables p ++ variables q
```

- `(interpretacionesVar n)` es la lista de las interpretaciones con  $n$  variables. Por ejemplo,

```
*Main> interpretacionesVar 2
[[False,False],
 [False,True],
 [True,False],
 [True,True]]
```

```
interpretacionesVar :: Int -> [[Bool]]
interpretacionesVar 0 = [[]]
interpretacionesVar (n+1) =
  map (False:) bss ++ map (True:) bss
  where bss = interpretacionesVar n
```

- `(interpretaciones p)` es la lista de las interpretaciones de la fórmula  $p$ . Por ejemplo,

```
*Main> interpretaciones p3
[[('A',False),('B',False)],
 [('A',False),('B',True)],
 [('A',True),('B',False)],
 [('A',True),('B',True)]]
```

```
interpretaciones :: FProp -> [Interpretacion]
interpretaciones p =
  map (zip vs) (interpretacionesVar (length vs))
  where vs = nub (variables p)
```

### Decisión de tautología

- `(esTautologia p)` se verifica si la fórmula  $p$  es una tautología. Por ejemplo,

```
esTautologia p1 ~> False
esTautologia p2 ~> True
esTautologia p3 ~> False
esTautologia p4 ~> True
```

```
esTautologia :: FProp -> Bool
esTautologia p =
  and [valor i p | i <- interpretaciones p]
```

## 9.5. Máquina abstracta de cálculo aritmético

### Evaluación de expresiones aritméticas

- Una expresión aritmética es un número entero o la suma de dos expresiones.

```
data Expr = Num Int | Suma Expr Expr
```

- `(valorEA x)` es el valor de la expresión aritmética `x`.

```
valorEA (Suma (Suma (Num 2) (Num 3)) (Num 4)) ~> 9
```

```
valorEA :: Expr -> Int
valorEA (Num n)      = n
valorEA (Suma x y) = valorEA x + valorEA y
```

- Cálculo:

```
valorEA (Suma (Suma (Num 2) (Num 3)) (Num 4))
= (valorEA (Suma (Num 2) (Num 3))) + (valorEA (Num 4))
= (valorEA (Suma (Num 2) (Num 3))) + 4
= (valorEA (Num 2) + (valorEA (Num 3))) + 4
= (2 + 3) + 4
= 9
```

### Máquina de cálculo aritmético

- La pila de control de la máquina abstracta es una lista de operaciones.

```
type PControl = [Op]
```

- Las operaciones son meter una expresión en la pila o sumar un número con el primero de la pila.

```
data Op = METE Expr | SUMA Int
```

- `(eval x p)` evalúa la expresión `x` con la pila de control `p`. Por ejemplo,

```
eval (Suma (Suma (Num 2) (Num 3)) (Num 4)) [] ~> 9
eval (Suma (Num 2) (Num 3)) [METE (Num 4)] ~> 9
eval (Num 3) [SUMA 2, METE (Num 4)] ~> 9
eval (Num 4) [SUMA 5] ~> 9
```

```
eval :: Expr -> PControl -> Int
eval (Num n)    p = ejec p n
eval (Suma x y) p = eval x (METE y : p)
```

- (ejec p n) ejecuta la lista de control p sobre el entero n. Por ejemplo,

```
ejec [METE (Num 3), METE (Num 4)] 2 ~> 9
ejec [SUMA 2, METE (Num 4)]      3 ~> 9
ejec [METE (Num 4)]              5 ~> 9
ejec [SUMA 5]                    4 ~> 9
ejec []                          9 ~> 9
```

```
ejec :: PControl -> Int -> Int
ejec []          n = n
ejec (METE y : p) n = eval y (SUMA n : p)
ejec (SUMA n : p) m = ejec p (n+m)
```

- (evalua e) evalúa la expresión aritmética e con la máquina abstracta. Por ejemplo,

```
evalua (Suma (Suma (Num 2) (Num 3)) (Num 4)) ~> 9
```

```
evalua :: Expr -> Int
evalua e = eval e []
```

- Evaluación:

```
eval (Suma (Suma (Num 2) (Num 3)) (Num 4)) []
= eval (Suma (Num 2) (Num 3)) [METE (Num 4)]
= eval (Num 2) [METE (Num 3), METE (Num 4)]
= ejec [METE (Num 3), METE (Num 4)] 2
= eval (Num 3) [SUMA 2, METE (Num 4)]
= ejec [SUMA 2, METE (Num 4)] 3
= ejec [METE (Num 4)] (2+3)
= ejec [METE (Num 4)] 5
= eval (Num 4) [SUMA 5]
= ejec [SUMA 5] 4
= ejec [] (5+4)
= ejec [] 9
= 9
```

## 9.6. Declaraciones de clases y de instancias

### Declaraciones de clases

- Las clases se declaran mediante el mecanismo `class`.
- Ejemplo de declaración de clases:

```

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

  -- Minimal complete definition: (==) or (/=)
  x == y = not (x/=y)
  x /= y = not (x==y)

```

### Declaraciones de instancias

- Las instancias se declaran mediante el mecanismo `instance`.
- Ejemplo de declaración de instancia:

```

----- Prelude -----
instance Eq Bool where
  False == False = True
  True  == True  = True
  _     == _     = False

```

### Extensiones de clases

- Las clases pueden extenderse mediante el mecanismo `class`.
- Ejemplo de extensión de clases:

```

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

  -- Minimal complete definition: (<=) or compare
  -- using compare can be more efficient for complex types
  compare x y | x==y      = EQ
              | x<=y      = LT

```

```

        | otherwise = GT

x <= y      = compare x y /= GT
x < y       = compare x y == LT
x >= y      = compare x y /= LT
x > y       = compare x y == GT

max x y    | x <= y      = y
           | otherwise  = x
min x y    | x <= y      = x
           | otherwise  = y

```

### Instancias de clases extendidas

- Las instancias de las clases extendidas pueden declararse mediante el mecanismo `instance`.
- Ejemplo de declaración de instancia:

```

----- Prelude -----
instance Ord Bool where
  False <= _      = True
  True  <= True   = True
  True  <= False  = False

```

### Clases derivadas

- Al definir un nuevo tipo con `data` puede declararse como instancia de clases mediante el mecanismo `deriving`.
- Ejemplo de clases derivadas:

```

----- Prelude -----
data Bool = False | True
          deriving (Eq, Ord, Read, Show)

```

- Comprobación:

```

False == False      ~> True
False < True         ~> True
show False          ~> "False"
read "False" :: Bool ~> False

```



- Para derivar un tipo cuyos constructores tienen argumentos como derivado, los tipos de los argumentos tienen que ser instancias de las clases derivadas.
- Ejemplo:

```
data Figura = Circulo Float | Rect Float Float
            deriving (Eq, Ord, Show)
```

se cumple que Float es instancia de Eq, Ord y Show.

```
*Main> :info Float
...
instance Eq Float
instance Ord Float
instance Show Float
...
```

## Bibliografía

1. G. Hutton *Programming in Haskell*. Cambridge University Press, 2007.
  - Cap. 10: Declaring types and classes.
2. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. *Razonando con Haskell*. Thompson, 2004.
  - Cap. 4: Definición de tipos.
  - Cap. 5: El sistema de clases de Haskell.
3. S. Thompson. *Haskell: The Craft of Functional Programming*, Second Edition. Addison-Wesley, 1999.
  - Cap. 12: Overloading and type classes.
  - Cap. 13: Checking types.
  - Cap. 14: Algebraic types.



# Tema 10

## Evaluación perezosa

### Contenido

---

<b>10.1. Estrategias de evaluación</b> . . . . .	<b>109</b>
<b>10.2. Terminación</b> . . . . .	<b>110</b>
<b>10.3. Número de reducciones</b> . . . . .	<b>111</b>
<b>10.4. Estructuras infinitas</b> . . . . .	<b>112</b>
<b>10.5. Programación modular</b> . . . . .	<b>113</b>
<b>10.6. Aplicación estricta</b> . . . . .	<b>114</b>

---

### 10.1. Estrategias de evaluación

#### Estrategias de evaluación

- Para los ejemplos se considera la función

```
mult :: (Int,Int) -> Int
mult (x,y) = x*y
```

- Evaluación mediante paso de parámetros por valor (o por más internos):
  - mult (1+2,2+3)
  - = mult (3,5)      [por def. de +]
  - = 3\*5            [por def. de mult]
  - = 15             [por def. de \*]

- Evaluación mediante paso de parámetros por nombre (o por más externos):

```

mult (1+2,2+3)
= (1+2)*(3+5)    [por def. de mult]
= 3*5            [por def. de +]
= 15             [por def. de *]

```

### Evaluación con lambda expresiones

- Se considera la función

```

mult' :: Int -> Int -> Int
mult' x = \y -> x*y

```

- Evaluación:

```

mult' (1+2) (2+3)
= mult' 3 (2+3)    [por def. de +]
= (\y -> 3*y) (2+3) [por def. de mult']
= (\y -> 3*y) 5    [por def. de +]
= 3*5              [por def. de +]
= 15               [por def. de *]

```

## 10.2. Terminación

### Procesamiento con el infinito

- Definición de infinito

```

inf :: Int
inf = 1 + inf

```

- Evaluación de infinito en Haskell:

```

*Main> inf
C-c C-c Interrupted.

```

- Evaluación de infinito:

```

inf
= 1 + inf          [por def. inf]
= 1 + (1 + inf)   [por def. inf]
= 1 + (1 + (1 + inf)) [por def. inf]
= ...

```

### Procesamiento con el infinito

- Evaluación mediante paso de parámetros por valor:

```
fst (0,inf)
= fst (0,1 + inf)      [por def. inf]
= fst (0,1 + (1 + inf)) [por def. inf]
= fst (0,1 + (1 + (1 + inf))) [por def. inf]
= ...
```

- Evaluación mediante paso de parámetros por nombre:

```
fst (0,inf)
= 0      [por def. fst]
```

- Evaluación Haskell con infinito:

```
*Main> fst (0,inf)
0
```

## 10.3. Número de reducciones

### Número de reducciones según las estrategias

- Para los ejemplos se considera la función

```
cuadrado :: Int -> Int
cuadrado n = n * n
```

- Evaluación mediante paso de parámetros por valor:

```
cuadrado (1+2)
= cuadrado 3      [por def. +]
= 3*3             [por def. cuadrado]
= 9               [por def. de *]
```

- Evaluación mediante paso de parámetros por nombre:

```
cuadrado (1+2)
= (1+2)*(1+2)    [por def. cuadrado]
= 3*(1+2)        [por def. de +]
= 3*3            [por def. de +]
= 9              [por def. de *]
```

### Evaluación perezosa e impaciente

- En la evaluación mediante paso de parámetros por nombre los argumentos pueden evaluarse más veces que en el paso por valor.
- Se puede usar punteros para compartir valores de expresiones.
- La evaluación mediante paso de parámetros por nombre usando punteros para compartir valores de expresiones se llama **evaluación perezosa**.
- La evaluación mediante paso de parámetros por valor se llama **evaluación impaciente**.
- Evaluación perezosa del ejemplo anterior:
  - cuadrado (1+2)
  - =  $x*x$  con  $x = 1+2$  [por def. cuadrado]
  - =  $3*3$  [por def. de +]
  - =  $9$  [por def. de \*]
- Haskell usa evaluación perezosa.

## 10.4. Estructuras infinitas

### Programación con estructuras infinitas

- unos es una lista infinita de unos.

```
unos :: [Int]
unos = 1 : unos
```

- Evaluación:
  - unos
  - =  $1 : \text{unos}$  [por def. unos]
  - =  $1 : (1 : \text{unos})$  [por def. unos]
  - =  $1 : (1 : (1 : \text{unos}))$  [por def. unos]
  - = ...

- Evaluación en Haskell:

```
*Main> unos
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,...
```

## Evaluación con estructuras infinitas

- Evaluación impaciente:

```

head unos
= head (1 : unos)           [por def. unos]
= head (1 : (1 : unos))    [por def. unos]
= head (1 : (1 : (1 : unos))) [por def. unos]
= ...

```

- Evaluación perezosa:

```

head unos
= head (1 : unos) [por def. unos]
= 1               [por def. head]

```

- Evaluación Haskell:

```

*Main> head unos
1

```

## 10.5. Programación modular

### Programación modular

- La evaluación perezosa permite separar el control de los datos.
- Para los ejemplos se considera la función

```

----- Prelude -----
take :: Int -> [a] -> [a]
take n _ | n <= 0 = []
take _ []         = []
take n (x:xs)    = x : take (n-1) xs

```

- Ejemplo de separación del control (tomar 2 elementos) de los datos (una lista infinita de unos):

```

take 2 unos
= take 2 (1 : unos)           [por def. unos]
= 1 : (take 1 unos)          [por def. take]
= 1 : (take 1 (1 : unos))    [por def. unos]
= 1 : (1 : (take 0 unos))    [por def. take]
= 1 : (1 : [])               [por def. take]
= [1,1]                       [por notación de listas]

```

### Terminación de evaluaciones con estructuras infinitas

- Ejemplo de no terminación:

```
*Main> [1..]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,...
```

- Ejemplo de terminación:

```
*Main> take 3 [1..]
[1,2,3]
```

- Ejemplo de no terminación:

```
*Main> filter (<=3) [1..]
[1,2,3 C-c C-c Interrupted.
```

- Ejemplo de no terminación:

```
*Main> takeWhile (<=3) [1..]
[1,2,3]
```

### La criba de Eratóstenes

- La criba de Eratóstenes

2	3	4	5	6	7	8	9	10	11	12	13	14	15	...
	3		5		7		9		11		13		15	...
		5		7					11		13			...
			7						11		13			...
									11		13			...
											13			...

- Definición

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

criba :: [Int] -> [Int]
criba (p:xs) = p : criba [x | x <- xs, x `mod` p /= 0]
```

- Evaluación:

```
take 15 primos ~> [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47]
```

- Cálculo:



```

primos
= criba [2..]
= criba (2 : [3..])
= 2 : (criba [x | x <- [3..], x 'mod' 2 /= 0])
= 2 : (criba (3 : [x | x <- [4..], x 'mod' 2 /= 0]))
= 2 : 3 : (criba [x | x <- [4..], x 'mod' 2 /= 0,
                x 'mod' 3 /= 0])
= 2 : 3 : (criba (5 : [x | x <- [6..], x 'mod' 2 /= 0,
                    x 'mod' 3 /= 0]))
= 2 : 3 : 5 : (criba ([x | x <- [6..], x 'mod' 2 /= 0,
                    x 'mod' 3 /= 0,
                    x 'mod' 5 /= 0]))
= ...

```

## 10.6. Aplicación estricta

### Ejemplo de programa sin aplicación estricta

- (`sumaNE xs`) es la suma de los números de `xs`. Por ejemplo,

```
| sumaNE [2,3,5]  ~>  10
```

```

sumaNE :: [Int] -> Int
sumaNE xs = sumaNE' 0 xs

sumaNE' :: Int -> [Int] -> Int
sumaNE' v []      = v
sumaNE' v (x:xs) = sumaNE' (v+x) xs

```

- Evaluación: :

```

sumaNE [2,3,5]
= sumaNE' 0 [2,3,5]      [por def. sumaNE]
= sumaNE' (0+2) [3,5]   [por def. sumaNE']
= sumaNE' ((0+2)+3) [5] [por def. sumaNE']
= sumaNE' (((0+2)+3)+5) [] [por def. sumaNE']
= ((0+2)+3)+5          [por def. sumaNE']
= (2+3)+5              [por def. +]
= 5+5                  [por def. +]
= 10                    [por def. +]

```

### Ejemplo de programa con aplicación estricta

- `(sumaE xs)` es la suma de los números de `xs`. Por ejemplo,

```
| sumaE [2,3,5] ~> 10
```

```
sumaE :: [Int] -> Int
sumaE xs = sumaE' 0 xs

sumaE' :: Int -> [Int] -> Int
sumaE' v [] = v
sumaE' v (x:xs) = (sumaE' $(v+x)) xs
```

- Evaluación: :

```
sumaE [2,3,5]
= sumaE' 0 [2,3,5]      [por def. sumaE]
= (sumaE' $(0+2)) [3,5] [por def. sumaE']
= sumaE' 2 [3,5]       [por aplicación de $!]
= (sumaE' $(2+3)) [5]  [por def. sumaE']
= sumaE' 5 [5]         [por aplicación de $!]
= (sumaE' $(5+5)) []   [por def. sumaE']
= sumaE' 10 []         [por aplicación de $!]
= 10                   [por def. sumaE']
```

### Comparación de consumo de memoria

- Comparación de consumo de memoria:

```
*Main> sumaNE [1..1000000]
*** Exception: stack overflow
*Main> sumaE [1..1000000]
1784293664
*Main> :set +s
*Main> sumaE [1..1000000]
1784293664
(2.16 secs, 145435772 bytes)
```

### Plegado estricto

- Versión estricta de `foldl` en el `Data.List`

```
foldl' :: (a -> b -> a) -> a -> [b] -> a
foldl' f a []      = a
foldl' f a (x:xs) = (foldl' f $! f a x) xs
```

- Comparación de plegado y plegado estricto:s

```
*Main> foldl (+) 0 [2,3,5]
10
*Main> foldl' (+) 0 [2,3,5]
10
*Main> foldl (+) 0 [1..1000000]
*** Exception: stack overflow
*Main> foldl' (+) 0 [1..1000000]
500000500000
```

## Bibliografía

1. R. Bird. *Introducción a la programación funcional con Haskell*. Prentice Hall, 2000.
  - Cap. 7: Eficiencia.
2. G. Hutton *Programming in Haskell*. Cambridge University Press, 2007.
  - Cap. 12: Lazy evaluation.
3. B. O'Sullivan, D. Stewart y J. Goerzen *Real World Haskell*. O'Reilly, 2008.
  - Cap. 2: Types and Functions.
4. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. *Razonando con Haskell*. Thomson, 2004.
  - Cap. 2: Introducción a Haskell.
  - Cap. 8: Evaluación perezosa. Redes de procesos.
5. S. Thompson. *Haskell: The Craft of Functional Programming*, Second Edition. Addison-Wesley, 1999.
  - Cap. 17: Lazy programming.



# Tema 11

## Analizadores sintácticos funcionales

### Contenido

---

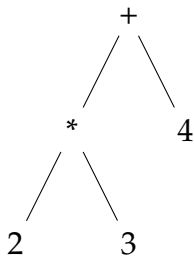
<b>11.1. Analizadores sintácticos</b> . . . . .	119
<b>11.2. El tipo de los analizadores sintácticos</b> . . . . .	119
<b>11.3. Analizadores sintácticos básicos</b> . . . . .	120
<b>11.4. Composición de analizadores sintácticos</b> . . . . .	121
11.4.1. Secuenciación de analizadores sintácticos . . . . .	121
11.4.2. Elección de analizadores sintácticos . . . . .	122
<b>11.5. Primitivas derivadas</b> . . . . .	122
<b>11.6. Tratamiento de los espacios</b> . . . . .	125
<b>11.7. Analizador de expresiones aritméticas</b> . . . . .	126

---

### 11.1. Analizadores sintácticos

#### Analizadores sintácticos

- Un **analizador sintáctico** es un programa que analiza textos para determinar su **estructura sintáctica**.
- Ejemplo de análisis sintáctico aritmético: La estructura sintáctica de la cadena "2\*3+4" es el árbol



- El análisis sintáctico forma parte del preprocesamiento en la mayoría de las aplicaciones reales.

## 11.2. El tipo de los analizadores sintácticos

### Opciones para el tipo de los analizadores sintácticos

- Opción inicial:

```
type Analizador = String -> Tree
```

- Con la parte no analizada:

```
type Analizador = String -> (Tree,String)
```

- Con todos los análisis:

```
type Analizador = String -> [(Tree,String)]
```

- Con estructuras arbitrarias:

```
type Analizador a = String -> [(a,String)]
```

- Simplificación: analizadores que fallan o sólo dan un análisis.

## 11.3. Analizadores sintácticos básicos

### Analizadores sintácticos básicos: resultado

- `(analiza a cs)` analiza la cadena `cs` mediante el analizador `a`. Por ejemplo,

```
analiza :: Analizador a -> String -> [(a,String)]
analiza a cs = a cs
```

- El analizador resultado `v` siempre tiene éxito, devuelve `v` y no consume nada. Por ejemplo,

```
*Main> analiza (resultado 1) "abc"
[(1,"abc")]
```

```
resultado :: a -> Analizador a
resultado v = \xs -> [(v,xs)]
```

### Analizadores sintácticos básicos: fallo

- El analizador fallo siempre falla. Por ejemplo,

```
*Main> analiza fallo "abc"
[]
```

```
fallo :: Analizador a
fallo = \xs -> []
```

### Analizadores sintácticos básicos: elemento

- El analizador elemento falla si la cadena es vacía y consume el primer elemento en caso contrario. Por ejemplo,

```
*Main> analiza elemento ""
[]
*Main> analiza elemento "abc"
[( 'a' ,"bc")]
```

```
elemento :: Analizador Char
elemento = \xs -> case xs of
    [] -> []
    (x:xs) -> [(x , xs)]
```

## 11.4. Composición de analizadores sintácticos

### 11.4.1. Secuenciación de analizadores sintácticos

- $((p \text{ 'liga' } f) \text{ } e)$  falla si el análisis de `e` por `p` falla, en caso contrario, se obtiene un valor (`v`) y una salida (`s`), se aplica la función `f` al valor `v` obteniéndose un nuevo analizador con el que se analiza la salida `s`.

```

liga :: Analizador a ->
      (a -> Analizador b) ->
      Analizador b
p 'liga' f = \ent -> case analiza p ent of
                    []      -> []
                    [(v,sal)] -> analiza (f v) sal

```

- primeroTercero es un analizador que devuelve los caracteres primero y tercero de la cadena. Por ejemplo,

```

primeroTercero "abel"  ~> [(('a','e'),"l")]
primeroTercero "ab"   ~> []

```

```

primeroTercero :: Analizador (Char,Char)
primeroTercero =
  elemento 'liga' \x ->
  elemento 'liga' \_ ->
  elemento 'liga' \y ->
  resultado (x,y)

```

### 11.4.2. Elección de analizadores sintácticos

- ((p +++ q) e) analiza e con p y si falla analiza e con q. Por ejemplo,

```

Main*> analiza (elemento +++ resultado 'd') "abc"
[(('a',"bc")]
Main*> analiza (fallo +++ resultado 'd') "abc"
[(('d',"abc")]
Main*> analiza (fallo +++ fallo) "abc"
[]

```

```

(+++) :: Analizador a -> Analizador a -> Analizador a
p +++ q = \ent -> case analiza p ent of
              []      -> analiza q ent
              [(v,sal)] -> [(v,sal)]

```

## 11.5. Primitivas derivadas



- `(sat p)` es el analizador que consume un elemento si dicho elemento cumple la propiedad `p` y falla en caso contrario. Por ejemplo,

```
analiza (sat isLower) "hola" ~> [('h',"ola")]
analiza (sat isLower) "Hola" ~> []
```

```
sat :: (Char -> Bool) -> Analizador Char
sat p = elemento 'liga' \x ->
    if p x then resultado x else fallo
```

- `digito` analiza si el primer carácter es un dígito. Por ejemplo,

```
analiza digito "123" ~> [('1',"23")]
analiza digito "uno" ~> []
```

```
digito :: Analizador Char
digito = sat isDigit
```

- `minuscula` analiza si el primer carácter es una letra minúscula. Por ejemplo,

```
analiza minuscula "eva" ~> [('e',"va")]
analiza minuscula "Eva" ~> []
```

```
minuscula :: Analizador Char
minuscula = sat isLower
```

- `mayuscula` analiza si el primer carácter es una letra mayúscula. Por ejemplo,

```
analiza mayuscula "Eva" ~> [('E',"va")]
analiza mayuscula "eva" ~> []
```

```
mayuscula :: Analizador Char
mayuscula = sat isUpper
```

- `letra` analiza si el primer carácter es una letra. Por ejemplo,

```
analiza letra "Eva" ~> [('E',"va")]
analiza letra "eva" ~> [('e',"va")]
analiza letra "123" ~> []
```

```
letra :: Analizador Char
letra = sat isAlpha
```

- `alfanumerico` analiza si el primer carácter es una letra o un número. Por ejemplo,

```
analiza alfanumerico "Eva"  ~> [('E',"va")]
analiza alfanumerico "eva"  ~> [('e',"va")]
analiza alfanumerico "123"  ~> [('1',"23")]
analiza alfanumerico " 123" ~> []
```

```
alfanumerico :: Analizador Char
alfanumerico = sat isAlphaNum
```

- `(caracter x)` analiza si el primer carácter es igual al carácter `x`. Por ejemplo,

```
analiza (caracter 'E') "Eva" ~> [('E',"va")]
analiza (caracter 'E') "eva" ~> []
```

```
caracter :: Char -> Analizador Char
caracter x = sat (== x)
```

- `(cadena c)` analiza si empieza con la cadena `c`. Por ejemplo,

```
analiza (cadena "abc") "abcdef" ~> [("abc","def")]
analiza (cadena "abc") "abdcef" ~> []
```

```
cadena :: String -> Analizador String
cadena []      = resultado []
cadena (x:xs) = caracter x 'liga' \x ->
                  cadena xs 'liga' \xs ->
                  resultado (x:xs)
```

- `varios p` aplica el analizador `p` cero o más veces. Por ejemplo,

```
analiza (varios digito) "235abc" ~> [("235","abc")]
analiza (varios digito) "abc235" ~> [("", "abc235")]
```

```
varios :: Analizador a -> Analizador [a]
varios p = varios1 p +++ resultado []
```

- `varios1 p` aplica el analizador `p` una o más veces. Por ejemplo,

```
analiza (varios1 digito) "235abc" ~> [("235","abc")]
analiza (varios1 digito) "abc235" ~> []
```

```
varios1 :: Analizador a -> Analizador [a]
varios1 p = p          'liga' \v ->
            varios p 'liga' \vs ->
            resultado (v:vs)
```

- `ident` analiza si comienza con un identificador (i.e. una cadena que comienza con una letra minúscula seguida por caracteres alfanuméricos). Por ejemplo,

```
Main*> analiza ident "lunes12 de Ene"
[("lunes12"," de Ene")]
Main*> analiza ident "Lunes12 de Ene"
[]
```

```
ident :: Analizador String
ident = minuscula      'liga' \x ->
        varios alfanumerico 'liga' \xs ->
        resultado (x:xs)
```

- `nat` analiza si comienza con un número natural. Por ejemplo,

```
analiza nat "14DeAbril" ~> [(14,"DeAbril")]
analiza nat " 14DeAbril" ~> []
```

```
nat :: Analizador Int
nat = varios1 digito 'liga' \xs ->
        resultado (read xs)
```

- `espacio` analiza si comienza con espacios en blanco. Por ejemplo,

```
analiza espacio "   a b c" ~> [((), "a b c")]
```

```
espacio :: Analizador ()
espacio = varios (sat isSpace) 'liga' \_ ->
        resultado ()
```

## 11.6. Tratamiento de los espacios

- `unidad p` ignora los espacios en blanco y aplica el analizador `p`. Por ejemplo,

```
Main*> analiza (unidad nat) " 14DeAbril"
[(14,"DeAbril")]
Main*> analiza (unidad nat) " 14  DeAbril"
[(14,"DeAbril")]
```

```
unidad :: Analizador a -> Analizador a
unidad p = espacio 'liga' \_ ->
           p          'liga' \v ->
           espacio 'liga' \_ ->
           resultado v
```

- `identificador` analiza un identificador ignorando los espacios delante y detrás. Por ejemplo,

```
Main*> analiza identificador "  lunes12  de Ene"
[("lunes12","de Ene")]
```

```
identificador :: Analizador String
identificador = unidad ident
```

- `natural` analiza un número natural ignorando los espacios delante y detrás. Por ejemplo,

```
analiza natural " 14DeAbril" ~> [(14,"DeAbril")]
```

```
natural :: Analizador Int
natural = unidad nat
```

- `(simbolo xs)` analiza la cadena `xs` ignorando los espacios delante y detrás. Por ejemplo,

```
Main*> analiza (simbolo "abc") " abcdef"
[("abc","def")]
```

```
simbolo :: String -> Analizador String
simbolo xs = unidad (cadena xs)
```

- `listaNat` analiza una lista de naturales ignorando los espacios. Por ejemplo,

```
Main*> analiza listaNat " [ 2, 3, 5  ]"
[[[2,3,5],""]]
Main*> analiza listaNat " [ 2, 3,]"
[]
```

```
listaNat :: Analizador [Int]
listaNat = simbolo "["          'liga' \_ ->
           natural              'liga' \n ->
           varios (simbolo ",", 'liga' \_ ->
                  natural)     'liga' \ns ->
           simbolo "]"         'liga' \_ ->
           resultado (n:ns)
```

## 11.7. Analizador de expresiones aritméticas

### Expresiones aritméticas

- Consideramos expresiones aritméticas:
  - construidas con números, operaciones (+ y \*) y paréntesis.
  - + y \* asocian por la derecha.
  - \* tiene más prioridad que +.
- Ejemplos:
  - $2 + 3 + 5$  representa a  $2 + (3 + 5)$ .
  - $2 * 3 + 5$  representa a  $(2 * 3) + 5$ .

### Gramáticas de las expresiones aritméticas: Gramática 1

- Gramática 1 de las expresiones aritméticas:
 
$$\begin{aligned} \text{expr} &::= \text{expr} + \text{expr} \mid \text{expr} * \text{expr} \mid (\text{expr}) \mid \text{nat} \\ \text{nat} &::= 0 \mid 1 \mid 2 \mid \dots \end{aligned}$$
- La gramática 1 no considera prioridad:
 

acepta  $2 + 3 * 5$  como  $(2 + 3) * 5$  y como  $2 + (3 * 5)$
- La gramática 1 no considera asociatividad:
 

acepta  $2 + 3 + 5$  como  $(2 + 3) + 5$  y como  $2 + (3 + 5)$
- La gramática 1 es ambigua.

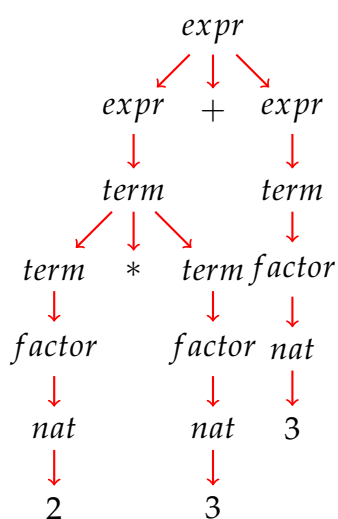
### Gramáticas de las expresiones aritméticas: Gramática 2

- Gramática 2 de las expresiones aritméticas (con prioridad):

$$\begin{aligned} \text{expr} &::= \text{expr} + \text{expr} \mid \text{term} \\ \text{term} &::= \text{term} * \text{term} \mid \text{factor} \\ \text{factor} &::= (\text{expr}) \mid \text{nat} \\ \text{nat} &::= 0 \mid 1 \mid 2 \mid \dots \end{aligned}$$

- La gramática 2 sí considera prioridad:  
acepta  $2 + 3 * 5$  sólo como  $2 + (3 * 5)$
- La gramática 2 no considera asociatividad:  
acepta  $2 + 3 + 5$  como  $(2 + 3) + 5$  y como  $2 + (3 + 5)$
- La gramática 2 es ambigua.

### Árbol de análisis sintáctico de $2 * 3 + 5$ con la gramática 2



### Gramáticas de las expresiones aritméticas: Gramática 3

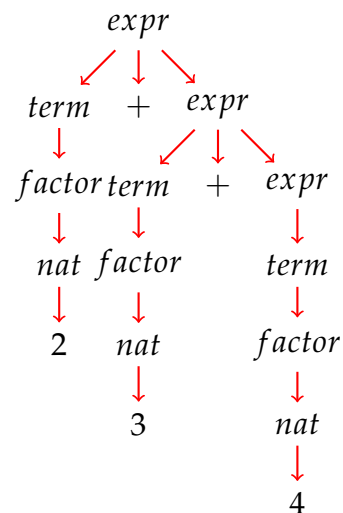
- Gramática 3 de las expresiones aritméticas:

$$\begin{aligned} \text{expr} &::= \text{term} + \text{expr} \mid \text{term} \\ \text{term} &::= \text{factor} * \text{term} \mid \text{factor} \\ \text{factor} &::= (\text{expr}) \mid \text{nat} \\ \text{nat} &::= 0 \mid 1 \mid 2 \mid \dots \end{aligned}$$

- La gramática 3 sí considera prioridad:  
acepta  $2 + 3 * 5$  sólo como  $2 + (3 * 5)$

- La gramática 3 sí considera asociatividad:  
acepta  $2 + 3 + 5$  como  $2 + (3 + 5)$
- La gramática 3 no es ambigua (i.e. es libre de contexto).

### Árbol de análisis sintáctico de $2 + 3 + 5$ con la gramática 3



### Gramáticas de las expresiones aritméticas: Gramática 4

- La gramática 4 se obtiene simplificando la gramática 3:

$$\begin{aligned}
 \text{expr} &::= \text{term} (+ \text{expr} \mid \epsilon) \\
 \text{term} &::= \text{factor} (* \text{term} \mid \epsilon) \\
 \text{factor} &::= (\text{expr}) \mid \text{nat} \\
 \text{nat} &::= 0 \mid 1 \mid 2 \mid \dots
 \end{aligned}$$

donde  $\epsilon$  es la cadena vacía.

- La gramática 4 no es ambigua.
- La gramática 4 es la que se usará para escribir el analizador de expresiones aritméticas.

### Analizador de expresiones aritméticas

- `expr` analiza una expresión aritmética devolviendo su valor. Por ejemplo,

analiza expr "2*3+5"	$\rightsquigarrow$	[(11, "")]
analiza expr "2*(3+5)"	$\rightsquigarrow$	[(16, "")]
analiza expr "2+3*5"	$\rightsquigarrow$	[(17, "")]
analiza expr "2*3+5abc"	$\rightsquigarrow$	[(11, "abc")]

```

expr :: Analizador Int
expr = term          'liga' \t ->
      (simbolo "+"   'liga' \_ ->
        expr         'liga' \e ->
         resultado (t+e))
      +++ resultado t

```

- averbterm analiza un término de una expresión aritmética devolviendo su valor. Por ejemplo,

```

analiza term "2*3+5"    ~> [(6,"+5")]
analiza term "2+3*5"    ~> [(2,"+3*5")]
analiza term "(2+3)*5+7" ~> [(25,"+7")]

```

```

term :: Analizador Int
term = factor        'liga' \f ->
      (simbolo "*"   'liga' \_ ->
        term         'liga' \t ->
         resultado (f*t))
      +++ resultado f

```

- averbfactor analiza un factor de una expresión aritmética devolviendo su valor. Por ejemplo,

```

analiza factor "2*3+5"    ~> [(2,"*3+5")]
analiza factor "(2+3)*5"  ~> [(5,"*5")]
analiza factor "(2+3*7)*5" ~> [(23,"*5")]

```

```

factor :: Analizador Int
factor = (simbolo "(" 'liga' \_ ->
         expr         'liga' \e ->
         simbolo ")"  'liga' \_ ->
         resultado e)
         +++ natural

```

- (valor cs) analiza la cadena cs devolviendo su valor si es una expresión aritmética y un mensaje de error en caso contrario. Por ejemplo,

```

valor "2*3+5"    ~> 11
valor "2*(3+5)"  ~> 16

```



```
valor "2 * 3 + 5"  ~> 11
valor "2*3x"      ~> *** Exception: sin usar x
valor "-1"        ~> *** Exception: entrada no valida
```

```
valor :: String -> Int
valor xs = case (analiza expr xs) of
    [(n, [])] -> n
    [(_, sal)] -> error ("sin usar " ++ sal)
    []         -> error "entrada no valida"
```

## Bibliografía

1. R. Bird. *Introducción a la programación funcional con Haskell*. Prentice Hall, 2000.
  - Cap. 11: Análisis sintáctico.
2. G. Hutton *Programming in Haskell*. Cambridge University Press, 2007.
  - Cap. 8: Functional parsers.
3. G. Hutton y E. Meijer. [Monadic Parser Combinators](#). Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.
4. G. Hutton y E. Meijer. [Monadic Parsing in Haskell](#). *Journal of Functional Programming*, 8(4): 437—444, 1998.
5. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. *Razonando con Haskell*. Thomson, 2004.
  - Cap. 14: Analizadores.



# Tema 12

## Programas interactivos

### Contenido

---

<b>12.1. Programas interactivos</b> . . . . .	133
<b>12.2. El tipo de las acciones de entrada/salida</b> . . . . .	134
<b>12.3. Acciones básicas</b> . . . . .	134
<b>12.4. Secuenciación</b> . . . . .	134
<b>12.5. Primitivas derivadas</b> . . . . .	135
<b>12.6. Ejemplos de programas interactivos</b> . . . . .	136
12.6.1. Juego de adivinación interactivo . . . . .	136
12.6.2. Calculadora aritmética . . . . .	137
12.6.3. El juego de la vida . . . . .	140

---

### 12.1. Programas interactivos

- Los programas por lote no interactúan con los usuarios durante su ejecución.
- Los programas interactivos durante su ejecución pueden leer datos del teclado y escribir resultados en la pantalla.
- Problema:
  - Los programas interactivos tienen efectos laterales.
  - Los programa Haskell no tiene efectos laterales.

### Ejemplo de programa interactivo

- Especificación: El programa pide una cadena y dice el número de caracteres que tiene.
- Ejemplo de sesión:

```
-- *Main> longitudCadena
-- Escribe una cadena: "Hoy es lunes"
-- La cadena tiene 14 caracteres
```

- Programa:

```
longitudCadena :: IO ()
longitudCadena = do putStr "Escribe una cadena: "
                   xs <- getLine
                   putStr "La cadena tiene "
                   putStr (show (length xs))
                   putStrLn " caracteres"
```

## 12.2. El tipo de las acciones de entrada/salida

- En Haskell se pueden escribir programas interactivos usando tipos que distingan las expresiones puras de las **acciones** impuras que tienen efectos laterales.
- `IO a` es el tipo de las acciones que devuelven un valor del tipo `a`.
- Ejemplos:
  - `IO Char` es el tipo de las acciones que devuelven un carácter.
  - `IO ()` es el tipo de las acciones que no devuelven ningún valor.

## 12.3. Acciones básicas

- `getChar :: IO Char`  
La acción `getChar` lee un carácter del teclado, lo muestra en la pantalla y lo devuelve como valor.
- `putChar :: c -> IO ()`  
La acción `putChar c` escribe el carácter `c` en la pantalla y no devuelve ningún valor.

- `return a -> IO a`  
La acción `return c` devuelve el valor `c` sin ninguna interacción.
- Ejemplo:

```
*Main> putChar 'b'
b*Main> it
()
```

## 12.4. Secuenciación

- Una sucesión de acciones puede combinarse en una acción compuesta mediante expresiones **do**.
- Ejemplo:

```
ejSecuenciacion :: IO (Char,Char)
ejSecuenciacion = do x <- getChar
                    getChar
                    y <- getChar
                    return (x,y)
```

Lee dos caracteres y devuelve el par formado por ellos. Por ejemplo,

```
*Main> ejSecuenciacion
b f
('b', 'f')
```

## 12.5. Primitivas derivadas

- Lectura de cadenas del teclado:

```
----- Prelude -----
getLine :: IO String
getLine = do x <- getChar
            if x == '\n' then return []
            else do xs <- getLine
                    return (x:xs)
```

- Escritura de cadenas en la pantalla:

```

Prelude
putStr :: String -> IO ()
putStr []      = return ()
putStr (x:xs) = do putChar x
                   putStr xs

```

- Escritura de cadenas en la pantalla y salto de línea:

```

Prelude
putStrLn :: String -> IO ()
putStrLn xs = do putStr xs
                 putChar '\n'

```

- Ejecución de una lista de acciones:

```

Prelude
sequence_ :: [IO a] -> IO ()
sequence_ []      = return ()
sequence_ (a:as) = do a
                     sequence_ as

```

Por ejemplo,

```

*Main> sequence_ [putStrLn "uno", putStrLn "dos"]
uno
dos
*Main> it
()

```

### Ejemplo de programa con primitivas derivadas

- Especificación: El programa pide una cadena y dice el número de caracteres que tiene.
- Ejemplo de sesión:

```

-- *Main> longitudCadena
-- Escribe una cadena: "Hoy es lunes"
-- La cadena tiene 14 caracteres

```

- Programa:

```
longitudCadena :: IO ()
longitudCadena = do putStr "Escribe una cadena: "
                   xs <- getLine
                   putStr "La cadena tiene "
                   putStr (show (length xs))
                   putStrLn " caracteres"
```

## 12.6. Ejemplos de programas interactivos

### 12.6.1. Juego de adivinación interactivo

- Descripción: El programa le pide al jugador humano que piense un número entre 1 y 100 y trata de adivinar el número que ha pensado planteándole conjeturas a las que el jugador humano responde con mayor, menor o exacto según que el número pensado sea mayor, menor o igual que el número conjeturado por la máquina.
- Ejemplo de sesión:

```
Main> juego
Piensa un numero entre el 1 y el 100.
Es 50? [mayor/menor/exacto] mayor
Es 75? [mayor/menor/exacto] menor
Es 62? [mayor/menor/exacto] mayor
Es 68? [mayor/menor/exacto] exacto
Fin del juego
```

- Programa:

```
juego :: IO ()
juego =
  do putStrLn "Piensa un numero entre el 1 y el 100."
     adivina 1 100
     putStrLn "Fin del juego"

adivina :: Int -> Int -> IO ()
adivina a b =
  do putStr ("Es " ++ show conjetura ++ "? [mayor/menor/exacto] ")
     s <- getLine
     case s of
       "mayor" -> adivina (conjetura+1) b
```

```

    "menor" -> adivina a (conjetura-1)
    "exacto" -> return ()
    _       -> adivina a b
  where
    conjetura = (a+b) `div` 2

```

## 12.6.2. Calculadora aritmética

### Acciones auxiliares

- Escritura de caracteres sin eco:

```

getCh :: IO Char
getCh = do hSetEcho stdin False
          c <- getChar
          hSetEcho stdin True
          return c

```

- Limpieza de la pantalla:

```

limpiaPantalla :: IO ()
limpiaPantalla = putStr "\ESC[2J"

```

- Escritura en una posición:

```

type Pos = (Int,Int)

irA :: Pos -> IO ()
irA (x,y) = putStr ("\ESC[" ++
                   show y ++ ";" ++ show x ++
                   "H")

escribeEn :: Pos -> String -> IO ()
escribeEn p xs = do irA p
                   putStr xs

```

### Calculadora

```

calculadora :: IO ()
calculadora = do limpiaPantalla

```



```

        escribeCalculadora
        limpiar

escribeCalculadora :: IO ()
escribeCalculadora =
    do limpiaPantalla
       sequence_ [escribeEn (1,y) xs
                  | (y,xs) <- zip [1..13] imagenCalculadora]
       putStrLn ""

```

```

imagenCalculadora :: [String]
imagenCalculadora = ["+-----+",
                    "|           |",
                    "+---+---+---+---+",
                    "| q | c | d | = |",
                    "+---+---+---+---+",
                    "| 1 | 2 | 3 | + |",
                    "+---+---+---+---+",
                    "| 4 | 5 | 6 | - |",
                    "+---+---+---+---+",
                    "| 7 | 8 | 9 | * |",
                    "+---+---+---+---+",
                    "| 0 | ( | ) | / |",
                    "+---+---+---+---+"]

```

Los primeros cuatro botones permiten escribir las órdenes:

- q para salir ('quit'),
- c para limpiar la agenda ('clear'),
- d para borrar un carácter ('delete') y
- = para evaluar una expresión.

Los restantes botones permiten escribir las expresiones.

```

limpiar :: IO ()
limpiar = calc ""

calc :: String -> IO ()

```

```

calc xs = do escribeEnPantalla xs
            c <- getCh
            if elem c botones
            then procesa c xs
            else do calc xs

escribeEnPantalla xs =
  do escribeEn (3,2) "          "
     escribeEn (3,2) (reverse (take 13 (reverse xs)))

```

```

botones :: String
botones = standard ++ extra
  where
    standard = "qcd=123+456-789*0()/"
    extra    = "QCD \ESC\BS\DEL\n"

```

```

procesa :: Char -> String -> IO ()
procesa c xs
  | elem c "qQ\ESC"    = salir
  | elem c "dD\BS\DEL" = borrar xs
  | elem c "=\n"       = evaluar xs
  | elem c "cC"        = limpiar
  | otherwise          = agregar c xs

```

```

salir :: IO ()
salir = irA (1,14)

borrar :: String -> IO ()
borrar "" = calc ""
borrar xs = calc (init xs)

evaluar :: String -> IO ()
evaluar xs = case analiza expr xs of
  [(n,"")] -> calc (show n)
  -         -> do calc xs

agregar :: Char -> String -> IO ()
agregar c xs = calc (xs ++ [c])

```

### 12.6.3. El juego de la vida

#### Descripción del juego de la vida

- El tablero del juego de la vida es una malla formada por cuadrados (“células”) que se pliega en todas las direcciones.
- Cada célula tiene 8 células vecinas, que son las que están próximas a ella, incluso en las diagonales.
- Las células tienen dos estados: están “vivas” o “muertas”.
- El estado del tablero evoluciona a lo largo de unidades de tiempo discretas.
- Las transiciones dependen del número de células vecinas vivas:
  - Una célula muerta con exactamente 3 células vecinas vivas “nace” (al turno siguiente estará viva).
  - Una célula viva con 2 ó 3 células vecinas vivas sigue viva, en otro caso muere.

#### El tablero del juego de la vida

- Tablero:

```
type Tablero = [Pos]
```

- Dimensiones:

```
ancho :: Int
ancho = 5

alto :: Int
alto = 5
```

#### El juego de la vida

- Ejemplo de tablero:

```
ejTablero :: Tablero
ejTablero = [(2,3), (3,4), (4,2), (4,3), (4,4)]
```

- Representación del tablero:

```

| 1234
| 1
| 2 0
| 3 0 0
| 4 00

```

- (vida n t) simula el juego de la vida a partir del tablero t con un tiempo entre generaciones proporcional a n. Por ejemplo,

```
| vida 100000 ejTablero
```

```

vida :: Int -> Tablero -> IO ()
vida n t = do limpiaPantalla
              escribeTablero t
              espera n
              vida n (siguienteGeneracion t)

```

- Escritura del tablero:

```

escribeTablero :: Tablero -> IO ()
escribeTablero t = sequence_ [escribeEn p "0" | p <- t]

```

- Espera entre generaciones:

```

espera :: Int -> IO ()
espera n = sequence_ [return () | _ <- [1..n]]

```

- siguienteGeneracion t) es el tablero de la siguiente generación al tablero t. Por ejemplo,

```

*Main> siguienteGeneracion ejTablero
| [(4,3),(3,4),(4,4),(3,2),(5,3)]

```

```

siguienteGeneracion :: Tablero -> Tablero
siguienteGeneracion t = supervivientes t ++ nacimientos t

```

- (supervivientes t) es la listas de posiciones de t que sobreviven; i.e. posiciones con 2 ó 3 vecinos vivos. Por ejemplo,

```
| supervivientes ejTablero ~> [(4,3),(3,4),(4,4)]
```

```

supervivientes :: Tablero -> [Pos]
supervivientes t = [p | p <- t,
                    elem (nVecinosVivos t p) [2,3]]

```

- `(nVecinosVivos t c)` es el número de vecinos vivos de la célula `c` en el tablero `t`. Por ejemplo,

```

nVecinosVivos ejTablero (3,3) ~> 5
nVecinosVivos ejTablero (3,4) ~> 3

```

```

nVecinosVivos :: Tablero -> Pos -> Int
nVecinosVivos t = length . filter (tieneVida t) . vecinos

```

`(vecinos p)` es la lista de los vecinos de la célula en la posición `p`. Por ejemplo,

```

vecinos (2,3) ~> [(1,2), (2,2), (3,2), (1,3), (3,3), (1,4), (2,4), (3,4)]
vecinos (1,2) ~> [(5,1), (1,1), (2,1), (5,2), (2,2), (5,3), (1,3), (2,3)]
vecinos (5,2) ~> [(4,1), (5,1), (1,1), (4,2), (1,2), (4,3), (5,3), (1,3)]
vecinos (2,1) ~> [(1,5), (2,5), (3,5), (1,1), (3,1), (1,2), (2,2), (3,2)]
vecinos (2,5) ~> [(1,4), (2,4), (3,4), (1,5), (3,5), (1,1), (2,1), (3,1)]
vecinos (1,1) ~> [(5,5), (1,5), (2,5), (5,1), (2,1), (5,2), (1,2), (2,2)]
vecinos (5,5) ~> [(4,4), (5,4), (1,4), (4,5), (1,5), (4,1), (5,1), (1,1)]

```

```

vecinos :: Pos -> [Pos]
vecinos (x,y) = map modular [(x-1,y-1), (x,y-1), (x+1,y-1),
                             (x-1,y),           (x+1,y),
                             (x-1,y+1), (x,y+1), (x+1,y+1)]

```

- `(modular p)` es la posición correspondiente a `p` en el tablero considerando los plegados. Por ejemplo,

```

modular (6,3) ~> (1,3)
modular (0,3) ~> (5,3)
modular (3,6) ~> (3,1)
modular (3,0) ~> (3,5)

```

```

modular :: Pos -> Pos
modular (x,y) = (((x-1) `mod` ancho) + 1, ((y-1) `mod` alto + 1))

```

- `(tieneVida t p)` se verifica si la posición `p` del tablero `t` tiene vida. Por ejemplo,

```
tieneVida ejTablero (1,1) ~> False
tieneVida ejTablero (2,3) ~> True
```

```
tieneVida :: Tablero -> Pos -> Bool
tieneVida t p = elem p t
```

- `(noTieneVida t p)` se verifica si la posición `p` del tablero `t` no tiene vida. Por ejemplo,

```
noTieneVida ejTablero (1,1) ~> True
noTieneVida ejTablero (2,3) ~> False
```

```
noTieneVida :: Tablero -> Pos -> Bool
noTieneVida t p = not (tieneVida t p)
```

- `(nacimientos t)` es la lista de los nacimientos de tablero `t`; i.e. las posiciones sin vida con 3 vecinos vivos. Por ejemplo,

```
nacimientos ejTablero ~> [(3,2), (5,3)]
```

```
nacimientos' :: Tablero -> [Pos]
nacimientos' t = [(x,y) | x <- [1..ancho],
                        y <- [1..alto],
                        noTieneVida t (x,y),
                        nVecinosVivos t (x,y) == 3]
```

- Definición más eficiente de `nacimientos`

```
nacimientos :: Tablero -> [Pos]
nacimientos t = [p | p <- nub (concat (map vecinos t)),
                 noTieneVida t p,
                 nVecinosVivos t p == 3]
```

donde `(nub xs)` es la lista obtenida eliminando las repeticiones de `xs`. Por ejemplo,

```
nub [2,3,2,5] ~> [2,3,5]
```

## Bibliografía

1. H. Daumé III. [Yet Another Haskell Tutorial](#). 2006.
  - Cap. 5: Basic Input/Output.
2. G. Hutton. *Programming in Haskell*. Cambridge University Press, 2007.
  - Cap. 9: Interactive programs.
3. B. O'Sullivan, J. Goerzen y D. Stewart. *Real World Haskell*. O'Reilly, 2009.
  - Cap. 7: I/O.
4. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. *Razonando con Haskell*. Thompson, 2004.
  - Cap. 7: Entrada y salida.
5. S. Thompson. *Haskell: The Craft of Functional Programming*, Second Edition. Addison-Wesley, 1999.
  - Cap. 18: Programming with actions.





# Tema 13

## Aplicaciones de programación funcional

### Contenido

---

<b>13.1. El juego de cifras y letras</b> . . . . .	145
13.1.1. Introducción . . . . .	145
13.1.2. Búsqueda de la solución por fuerza bruta . . . . .	149
13.1.3. Búsqueda combinando generación y evaluación . . . . .	151
13.1.4. Búsqueda mejorada mediante propiedades algebraicas . . . . .	153
<b>13.2. El problema de las reinas</b> . . . . .	156
<b>13.3. Números de Hamming</b> . . . . .	157

---

### 13.1. El juego de cifras y letras

#### 13.1.1. Introducción

##### Presentación del juego

- *Cifras y letras* es un programa de Canal Sur que incluye un juego numérico cuya esencia es la siguiente:

Dada una sucesión de números naturales y un número objetivo, intentar construir una expresión cuyo valor es el objetivo combinando los números de la sucesión usando suma, resta, multiplicación, división y paréntesis. Cada número de la sucesión puede usarse como máximo una vez. Además, todos los números, incluyendo los resultados intermedios tienen que ser enteros positivos (1,2,3,...).

- Ejemplos

- Dada la sucesión 1, 3, 7, 10, 25, 50 y el objetivo 765, una solución es  $(1+50)*(25-10)$ .
- Para el problema anterior, existen 780 soluciones.
- Con la sucesión anterior y el objetivo 831, no hay solución.

### Formalización del problema: Operaciones

- Las operaciones son sumar, restar, multiplicar o dividir.

```
data Op = Sum | Res | Mul | Div

instance Show Op where
  show Sum = "+"
  show Res = "-"
  show Mul = "*"
  show Div = "/"
```

- ops es la lista de las operaciones.

```
ops :: [Op]
ops = [Sum, Res, Mul, Div]
```

### Operaciones válidas

- (valida o x y) se verifica si la operación o aplicada a los números naturales x e y da un número natural. Por ejemplo,

```
valida Res 5 3 ~> True
valida Res 3 5 ~> False
valida Div 6 3 ~> True
valida Div 6 4 ~> False
```

```
valida :: Op -> Int -> Int -> Bool
valida Sum _ _ = True
valida Res x y = x > y
valida Mul _ _ = True
valida Div x y = x `mod` y == 0
```

## Aplicación de operaciones

- `(aplica o x y)` es el resultado de aplicar la operación `o` a los números naturales `x` e `y`. Por ejemplo,

```
aplica Sum 2 3 ~> 5
aplica Div 6 3 ~> 2
```

```
aplica :: Op -> Int -> Int -> Int
aplica Sum x y = x + y
aplica Res x y = x - y
aplica Mul x y = x * y
aplica Div x y = x `div` y
```

## Expresiones

- Las expresiones son números enteros o aplicaciones de operaciones a dos expresiones.

```
data Expr = Num Int | Apl Op Expr Expr

instance Show Expr where
  show (Num n)      = show n
  show (Apl o i d) = parenthesis i ++ show o ++ parenthesis d
    where
      parenthesis (Num n) = show n
      parenthesis e      = "(" ++ show e ++ ")"
```

- Ejemplo: Expresión correspondiente a  $(1+50)*(25-10)$

```
ejExpr :: Expr
ejExpr = Apl Mul e1 e2
  where e1 = Apl Sum (Num 1) (Num 50)
        e2 = Apl Res (Num 25) (Num 10)
```

## Números de una expresión

- `(numeros e)` es la lista de los números que aparecen en la expresión `e`. Por ejemplo,

```
*Main> numeros (Apl Mul (Apl Sum (Num 2) (Num 3)) (Num 7))
[2,3,7]
```

```

numeros :: Expr -> [Int]
numeros (Num n)      = [n]
numeros (Apl _ l r) = numeros l ++ numeros r

```

### Valor de una expresión

- (valor e) es la lista formada por el valor de la expresión e si todas las operaciones para calcular el valor de e son números positivos y la lista vacía en caso contrario. Por ejemplo,

```

valor (Apl Mul (Apl Sum (Num 2) (Num 3)) (Num 7)) ~> [35]
valor (Apl Res (Apl Sum (Num 2) (Num 3)) (Num 7)) ~> []
valor (Apl Sum (Apl Res (Num 2) (Num 3)) (Num 7)) ~> []

```

```

valor :: Expr -> [Int]
valor (Num n)      = [n | n > 0]
valor (Apl o i d) = [aplica o x y | x <- valor i
                                , y <- valor d
                                , valida o x y]

```

### Funciones combinatorias: Sublistas

- (sublistas xs) es la lista de las sublistas de xs. Por ejemplo,

```

*Main> sublistas "bc"
["","c","b","bc"]
*Main> sublistas "abc"
["","c","b","bc","a","ac","ab","abc"]

```

```

sublistas :: [a] -> [[a]]
sublistas []      = [[]]
sublistas (x:xs) = yss ++ map (x:) yss
  where yss = sublistas xs

```

### Funciones combinatoria: Intercalado

- (intercala x ys) es la lista de las listas obtenidas intercalando x entre los elementos de ys. Por ejemplo,

```
intercala 'x' "bc"  ~> ["xbc","bxc","bcx"]
intercala 'x' "abc" ~> ["xabc","axbc","abxc","abcx"]
```

```
intercala :: a -> [a] -> [[a]]
intercala x []      = [[x]]
intercala x (y:ys) =
  (x:y:ys) : map (y:) (intercala x ys)
```

### Funciones combinatoria: Permutaciones

- (permutaciones xs) es la lista de las permutaciones de xs. Por ejemplo,

```
*Main> permutaciones "bc"
["bc","cb"]
*Main> permutaciones "abc"
["abc","bac","bca","acb","cab","cba"]
```

```
permutaciones :: [a] -> [[a]]
permutaciones []      = [[]]
permutaciones (x:xs) =
  concat (map (intercala x) (permutaciones xs))
```

### Funciones combinatoria: Elecciones

- (elecciones xs) es la lista formada por todas las sublistas de xs en cualquier orden. Por ejemplo,

```
*Main> elecciones "abc"
["","c","b","bc","cb","a","ac","ca","ab","ba",
 "abc","bac","bca","acb","cab","cba"]
```

```
elecciones :: [a] -> [[a]]
elecciones xs =
  concat (map permutaciones (sublistas xs))
```

### Reconocimiento de las soluciones

- (solucion e ns n) se verifica si la expresión e es una solución para la sucesión ns y objetivo n; es decir. si los números de e es una posible elección de ns y el valor de e es n. Por ejemplo,

```
|solucion ejExpr [1,3,7,10,25,50] 765 => True
```

```
solucion :: Expr -> [Int] -> Int -> Bool
solucion e ns n =
    elem (numeros e) (elecciones ns) && valor e == [n]
```

### 13.1.2. Búsqueda de la solución por fuerza bruta

#### Divisiones de una lista

- (divisiones xs) es la lista de las divisiones de xs en dos listas no vacías. Por ejemplo,

```
*Main> divisiones "bcd"
[("b","cd"),("bc","d")]
*Main> divisiones "abcd"
[("a","bcd"),("ab","cd"),("abc","d")]
```

```
divisiones :: [a] -> [[a],[a]]
divisiones [] = []
divisiones [_] = []
divisiones (x:xs) =
    ([x],xs) : [(x:is,ds) | (is,ds) <- divisiones xs]
```

#### Expresiones construibles

- (expresiones ns) es la lista de todas las expresiones construibles a partir de la lista de números ns. Por ejemplo,

```
*Main> expresiones [2,3,5]
[2+(3+5),2-(3+5),2*(3+5),2/(3+5),2+(3-5),2-(3-5),
 2*(3-5),2/(3-5),2+(3*5),2-(3*5),2*(3*5),2/(3*5),
 2+(3/5),2-(3/5),2*(3/5),2/(3/5),(2+3)+5,(2+3)-5,
 ...
```

```
expresiones :: [Int] -> [Expr]
expresiones [] = []
expresiones [n] = [Num n]
expresiones ns = [e | (is,ds) <- divisiones ns
                    , i <- expresiones is
                    , d <- expresiones ds
                    , e <- combina i d]
```

### Combinación de expresiones

- `(combina e1 e2)` es la lista de las expresiones obtenidas combinando las expresiones `e1` y `e2` con una operación. Por ejemplo,

```
*Main> combina (Num 2) (Num 3)
[2+3,2-3,2*3,2/3]
```

```
combina :: Expr -> Expr -> [Expr]
combina e1 e2 = [Apl o e1 e2 | o <- ops]
```

### Búsqueda de las soluciones

- `(soluciones ns n)` es la lista de las soluciones para la sucesión `ns` y objetivo `n` calculadas por fuerza bruta. Por ejemplo,

```
*Main> soluciones [1,3,7,10,25,50] 765
[3*((7*(50-10))-25), ((7*(50-10))-25)*3, ...
*Main> length (soluciones [1,3,7,10,25,50] 765)
780
*Main> length (soluciones [1,3,7,10,25,50] 831)
0
```

```
soluciones :: [Int] -> Int -> [Expr]
soluciones ns n = [e | ns' <- elecciones ns
                    , e <- expresiones ns'
                    , valor e == [n]]
```

### Estadísticas de la búsqueda por fuerza bruta

- Estadísticas:

```
*Main> :set +s
*Main> head (soluciones [1,3,7,10,25,50] 765)
3*((7*(50-10))-25)
(8.47 secs, 400306836 bytes)
*Main> length (soluciones [1,3,7,10,25,50] 765)
780
(997.76 secs, 47074239120 bytes)
*Main> length (soluciones [1,3,7,10,25,50] 831)
0
(1019.13 secs, 47074535420 bytes)
*Main> :unset +s
```

### 13.1.3. Búsqueda combinando generación y evaluación

#### Resultados

- Resultado es el tipo de los pares formados por expresiones válidas y su valor.

```
type Resultado = (Expr, Int)
```

- (resultados ns) es la lista de todos los resultados construibles a partir de la lista de números ns. Por ejemplo,

```
*Main> resultados [2,3,5]
[(2+(3+5),10), (2*(3+5),16), (2+(3*5),17), (2*(3*5),30), ((2+3)+5,10),
((2+3)*5,25), ((2+3)/5,1), ((2*3)+5,11), ((2*3)-5,1), ((2*3)*5,30)]
```

```
resultados :: [Int] -> [Resultado]
resultados [] = []
resultados [n] = [(Num n,n) | n > 0]
resultados ns = [res | (is,ds) <- divisiones ns
                      , ix <- resultados is
                      , dy <- resultados ds
                      , res <- combina' ix dy]
```

#### Combinación de resultados

- (combina' r1 r2) es la lista de los resultados obtenidos combinando los resultados r1 y r2 con una operación. Por ejemplo,

```
*Main> combina' (Num 2,2) (Num 3,3)
[(2+3,5),(2*3,6)]
*Main> combina' (Num 3,3) (Num 2,2)
[(3+2,5),(3-2,1),(3*2,6)]
*Main> combina' (Num 2,2) (Num 6,6)
[(2+6,8),(2*6,12)]
*Main> combina' (Num 6,6) (Num 2,2)
[(6+2,8),(6-2,4),(6*2,12),(6/2,3)]
```

```
combina' :: Resultado -> Resultado -> [Resultado]
combina' (i,x) (d,y) =
  [(Apl o i d, aplica o x y) | o <- ops
                               , valida o x y]
```



### Búsqueda combinando generación y evaluación

- `(soluciones' ns n)` es la lista de las soluciones para la sucesión `ns` y objetivo `n` calculadas intercalando generación y evaluación. Por ejemplo,

```
*Main> head (soluciones' [1,3,7,10,25,50] 765)
3*((7*(50-10))-25)
*Main> length (soluciones' [1,3,7,10,25,50] 765)
780
*Main> length (soluciones' [1,3,7,10,25,50] 831)
0
```

```
soluciones' :: [Int] -> Int -> [Expr]
soluciones' ns n = [e | ns' <- elecciones ns
                      , (e,m) <- resultados ns'
                      , m == n]
```

### Estadísticas de la búsqueda combinada

- Estadísticas:

```
*Main> head (soluciones' [1,3,7,10,25,50] 765)
3*((7*(50-10))-25)
(0.81 secs, 38804220 bytes)
*Main> length (soluciones' [1,3,7,10,25,50] 765)
780
(60.73 secs, 2932314020 bytes)
*Main> length (soluciones' [1,3,7,10,25,50] 831)
0
(61.68 secs, 2932303088 bytes)
```

## 13.1.4. Búsqueda mejorada mediante propiedades algebraicas

### Aplicaciones válidas

- `(valida' o x y)` se verifica si la operación `o` aplicada a los números naturales `x` e `y` da un número natural, teniendo en cuenta las siguientes reducciones algebraicas

```
x + y = y + x
x * y = y * x
x * 1 = x
1 * y = y
x / 1 = x
```

```

valida' :: Op -> Int -> Int -> Bool
valida' Sum x y = x <= y
valida' Res x y = x > y
valida' Mul x y = x /= 1 && y /= 1 && x <= y
valida' Div x y = y /= 1 && x `mod` y == 0

```

### Resultados válidos construibles

- (resultados' ns) es la lista de todos los resultados válidos construibles a partir de la lista de números ns. Por ejemplo,

```

*Main> resultados' [5,3,2]
[(5-(3-2),4),((5-3)+2,4),((5-3)*2,4),((5-3)/2,1)]

```

```

resultados' :: [Int] -> [Resultado]
resultados' [] = []
resultados' [n] = [(Num n,n) | n > 0]
resultados' ns = [res | (is,ds) <- divisiones ns
                        , ix    <- resultados' is
                        , dy    <- resultados' ds
                        , res    <- combina'' ix dy]

```

### Combinación de resultados válidos

- (combina'' r1 r2) es la lista de los resultados válidos obtenidos combinando los resultados r1 y r2 con una operación. Por ejemplo,

```

combina'' (Num 2,2) (Num 3,3) => [(2+3,5),(2*3,6)]
combina'' (Num 3,3) (Num 2,2) => [(3-2,1)]
combina'' (Num 2,2) (Num 6,6) => [(2+6,8),(2*6,12)]
combina'' (Num 6,6) (Num 2,2) => [(6-2,4),(6/2,3)]

```

```

combina'' :: Resultado -> Resultado -> [Resultado]
combina'' (i,x) (d,y) =
  [(Apl o i d, aplica o x y) | o <- ops
                               , valida' o x y]

```

### Búsqueda mejorada mediante propiedades algebraicas

- (soluciones'' ns n) es la lista de las soluciones para la sucesión ns y objetivo n calculadas intercalando generación y evaluación y usando las mejoras aritméticas. Por ejemplo,

```
*Main> head (soluciones'' [1,3,7,10,25,50] 765)
3*((7*(50-10))-25)
*Main> length (soluciones'' [1,3,7,10,25,50] 765)
49
*Main> length (soluciones'' [1,3,7,10,25,50] 831)
0
```

```
soluciones'' :: [Int] -> Int -> [Expr]
soluciones'' ns n = [e | ns' <- elecciones ns
                      , (e,m) <- resultados' ns'
                      , m == n]
```

### Estadísticas de la búsqueda mejorada

- Estadísticas:

```
*Main> head (soluciones'' [1,3,7,10,25,50] 765)
3*((7*(50-10))-25)
(0.40 secs, 16435156 bytes)
*Main> length (soluciones'' [1,3,7,10,25,50] 765)
49
(10.30 secs, 460253716 bytes)
*Main> length (soluciones'' [1,3,7,10,25,50] 831)
0
(10.26 secs, 460253908 bytes)§
```

### Comparación de las búsquedas

Comparación de las búsquedas problema de dados [1,3,7,10,25,50] obtener 765.

- Búsqueda de la primera solución:

	segs.	bytes
soluciones	8.47	400.306.836
soluciones'	0.81	38.804.220

```
| soluciones'' | 0.40 | 16.435.156 |
+-----+-----+
```

### Comparación de las búsquedas

- Búsqueda de todas las soluciones:

```

          +-----+-----+
          | segs. | bytes |
+-----+-----+
| soluciones | 997.76 | 47.074.239.120 |
| soluciones' | 60.73 | 2.932.314.020 |
| soluciones'' | 10.30 | 460.253.716 |
+-----+-----+
```

### Comparación de las búsquedas

Comprobación de que dados [1,3,7,10,25,50] no puede obtenerse 831

```

          +-----+-----+
          | segs. | bytes |
+-----+-----+
| soluciones | 1019.13 | 47.074.535.420 |
| soluciones' | 61.68 | 2.932.303.088 |
| soluciones'' | 10.26 | 460.253.908 |
+-----+-----+
```

## 13.2. El problema de las reinas

### El problema de las N reinas

- Enunciado: Colocar N reinas en un tablero rectangular de dimensiones N por N de forma que no se encuentren más de una en la misma línea: horizontal, vertical o diagonal.
- El problema se representa en el módulo `Reinas`. Importa la diferencia de conjuntos (`\`) del módulo `List`:

```
module Reinas where
import Data.List ((\))
```

- El tablero se representa por una lista de números que indican las filas donde se han colocado las reinas. Por ejemplo,  $[3, 5]$  indica que se han colocado las reinas  $(1, 3)$  y  $(2, 5)$ .

```
type Tablero = [Int]
```

- `reinas n` es la lista de soluciones del problema de las  $N$  reinas. Por ejemplo, `reinas 4`  $\rightsquigarrow$   $[[3, 1, 4, 2], [2, 4, 1, 3]]$ . La primera solución  $[3, 1, 4, 2]$  se interpreta como

	R		
			R
R			
		R	

```
reinas :: Int -> [Tablero]
reinas n = aux n
  where aux 0      = [[]]
        aux (m+1) = [r:rs | rs <- aux m,
                          r <- ([1..n] \\ rs),
                          noAtaca r rs 1]
```

- `noAtaca r rs d` se verifica si la reina  $r$  no ataca a ninguna de las de la lista  $rs$  donde la primera de la lista está a una distancia horizontal  $d$ .

```
noAtaca :: Int -> Tablero -> Int -> Bool
noAtaca _ [] _ = True
noAtaca r (a:rs) distH = abs(r-a) /= distH &&
                          noAtaca r rs (distH+1)
```

## 13.3. Números de Hamming

### Números de Hamming

- Enunciado: Los números de Hamming forman una sucesión estrictamente creciente de números que cumplen las siguientes condiciones:
  - El número 1 está en la sucesión.
  - Si  $x$  está en la sucesión, entonces  $2x$ ,  $3x$  y  $5x$  también están.
  - Ningún otro número está en la sucesión.

- hamming es la sucesión de Hamming. Por ejemplo,

```
| take 12 hamming ~> [1,2,3,4,5,6,8,9,10,12,15,16]
```

```
hamming :: [Int]
hamming = 1 : mezcla3 [2*i | i <- hamming]
                  [3*i | i <- hamming]
                  [5*i | i <- hamming]
```

- mezcla3 xs ys zs es la lista obtenida mezclando las listas ordenadas xs, ys y zs y eliminando los elementos duplicados. Por ejemplo,

```
| Main> mezcla3 [2,4,6,8,10] [3,6,9,12] [5,10]
|[2,3,4,5,6,8,9,10,12]
```

```
mezcla3 :: [Int] -> [Int] -> [Int] -> [Int]
mezcla3 xs ys zs = mezcla2 xs (mezcla2 ys zs)
```

- mezcla2 xs ys es la lista obtenida mezclando las listas ordenadas xs e ys y eliminando los elementos duplicados. Por ejemplo,

```
| Main> mezcla2 [2,4,6,8,10,12] [3,6,9,12]
|[2,3,4,6,8,9,10,12]
```

```
mezcla2 :: [Int] -> [Int] -> [Int]
mezcla2 p@(x:xs) q@(y:ys) | x < y      = x:mezcla2 xs q
                          | x > y      = y:mezcla2 p ys
                          | otherwise = x:mezcla2 xs ys
mezcla2 []      ys          = ys
mezcla2 xs      []         = xs
```

## Bibliografía

1. G. Hutton *Programming in Haskell*. Cambridge University Press, 2007.
  - Cap. 11: The countdown problem.
2. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. *Razonando con Haskell*. Thomson, 2004.
  - Cap. 13: Puzzles y solitarios.

# Apéndice A

## Resumen de funciones predefinidas de Haskell

1. `x + y` es la suma de x e y.
2. `x - y` es la resta de x e y.
3. `x / y` es el cociente de x entre y.
4. `x ^ y` es x elevado a y.
5. `x == y` se verifica si x es igual a y.
6. `x /= y` se verifica si x es distinto de y.
7. `x < y` se verifica si x es menor que y.
8. `x <= y` se verifica si x es menor o igual que y.
9. `x > y` se verifica si x es mayor que y.
10. `x >= y` se verifica si x es mayor o igual que y.
11. `x && y` es la conjunción de x e y.
12. `x || y` es la disyunción de x e y.
13. `x:ys` es la lista obtenida añadiendo x al principio de ys.
14. `xs ++ ys` es la concatenación de xs e ys.
15. `xs !! n` es el elemento n-ésimo de xs.
16. `f . g` es la composición de f y g.
17. `abs x` es el valor absoluto de x.
18. `and xs` es la conjunción de la lista de booleanos xs.
19. `ceiling x` es el menor entero no menor que x.
20. `chr n` es el carácter cuyo código ASCII es n.
21. `concat xss` es la concatenación de la lista de listas xss.
22. `const x y` es x.

23. `curry f` es la versión curryficada de la función  $f$ .
24. `div x y` es la división entera de  $x$  entre  $y$ .
25. `drop n xs` borra los  $n$  primeros elementos de  $xs$ .
26. `dropWhile p xs` borra el mayor prefijo de  $xs$  cuyos elementos satisfacen el predicado  $p$ .
27. `elem x ys` se verifica si  $x$  pertenece a  $ys$ .
28. `even x` se verifica si  $x$  es par.
29. `filter p xs` es la lista de elementos de la lista  $xs$  que verifican el predicado  $p$ .
30. `flip f x y` es  $f y x$ .
31. `floor x` es el mayor entero no mayor que  $x$ .
32. `foldl f e xs` pliega  $xs$  de izquierda a derecha usando el operador  $f$  y el valor inicial  $e$ .
33. `foldr f e xs` pliega  $xs$  de derecha a izquierda usando el operador  $f$  y el valor inicial  $e$ .
34. `fromIntegral x` transforma el número entero  $x$  al tipo numérico correspondiente.
35. `fst p` es el primer elemento del par  $p$ .
36. `gcd x y` es el máximo común divisor de  $x$  e  $y$ .
37. `head xs` es el primer elemento de la lista  $xs$ .
38. `init xs` es la lista obtenida eliminando el último elemento de  $xs$ .
39. `isSpace x` se verifica si  $x$  es un espacio.
40. `isUpper x` se verifica si  $x$  está en mayúscula.
41. `isLower x` se verifica si  $x$  está en minúscula.
42. `isAlpha x` se verifica si  $x$  es un carácter alfabético.
43. `isDigit x` se verifica si  $x$  es un dígito.
44. `isAlphaNum x` se verifica si  $x$  es un carácter alfanumérico.
45. `iterate f x` es la lista  $[x, f(x), f(f(x)), \dots]$ .
46. `last xs` es el último elemento de la lista  $xs$ .
47. `length xs` es el número de elementos de la lista  $xs$ .
48. `map f xs` es la lista obtenida aplicado  $f$  a cada elemento de  $xs$ .
49. `max x y` es el máximo de  $x$  e  $y$ .
50. `maximum xs` es el máximo elemento de la lista  $xs$ .
51. `min x y` es el mínimo de  $x$  e  $y$ .
52. `minimum xs` es el mínimo elemento de la lista  $xs$ .
53. `mod x y` es el resto de  $x$  entre  $y$ .
54. `not x` es la negación lógica del booleano  $x$ .



55. `noElem x ys` se verifica si  $x$  no pertenece a  $ys$ .
56. `null xs` se verifica si  $xs$  es la lista vacía.
57. `odd x` se verifica si  $x$  es impar.
58. `or xs` es la disyunción de la lista de booleanos  $xs$ .
59. `ord c` es el código ASCII del carácter  $c$ .
60. `product xs` es el producto de la lista de números  $xs$ .
61. `rem x y` es el resto de  $x$  entre  $y$ .
62. `repeat x` es la lista infinita  $[x, x, x, \dots]$ .
63. `replicate n x` es la lista formada por  $n$  veces el elemento  $x$ .
64. `reverse xs` es la inversa de la lista  $xs$ .
65. `round x` es el redondeo de  $x$  al entero más cercano.
66. `scanr f e xs` es la lista de los resultados de plegar  $xs$  por la derecha con  $f$  y  $e$ .
67. `show x` es la representación de  $x$  como cadena.
68. `signum x` es 1 si  $x$  es positivo, 0 si  $x$  es cero y -1 si  $x$  es negativo.
69. `snd p` es el segundo elemento del par  $p$ .
70. `splitAt n xs` es  $(\text{take } n \text{ } xs, \text{drop } n \text{ } xs)$ .
71. `sqrt x` es la raíz cuadrada de  $x$ .
72. `sum xs` es la suma de la lista numérica  $xs$ .
73. `tail xs` es la lista obtenida eliminando el primer elemento de  $xs$ .
74. `take n xs` es la lista de los  $n$  primeros elementos de  $xs$ .
75. `takeWhile p xs` es el mayor prefijo de  $xs$  cuyos elementos satisfacen el predicado  $p$ .
76. `uncurry f` es la versión cartesiana de la función  $f$ .
77. `until p f x` aplica  $f$  a  $x$  hasta que se verifique  $p$ .
78. `zip xs ys` es la lista de pares formado por los correspondientes elementos de  $xs$  e  $ys$ .
79. `zipWith f xs ys` se obtiene aplicando  $f$  a los correspondientes elementos de  $xs$  e  $ys$ .



# Bibliografía

- [1] Richard Bird: [Introducción a la programación funcional con Haskell](#). (Prentice Hall, 2000).
- [2] Antony Davie: *An Introduction to Functional Programming Systems Using Haskell*. (Cambridge University Press, 1992).
- [3] Paul Hudak: *The Haskell School of Expression: Learning Functional Programming through Multimedia*. (Cambridge University Press, 2000).
- [4] Graham Hutton: [Programming in Haskell](#). (Cambridge University Press, 2007).
- [5] Bryan O’Sullivan, Don Stewart y John Goerzen: [Real World Haskell](#). (O’Reilly, 2008).
- [6] Blas C. Ruiz, Francisco Gutiérrez, Pablo Guerrero y José E. Gallardo: *Razonando con Haskell*. (Thompson, 2004).
- [7] Simon Thompson: *Haskell: The Craft of Functional Programming*, Second Edition. (Addison-Wesley, 1999).