

# Temas de “Programación declarativa” (2008–09)

José A. Alonso Jiménez

---

Grupo de Lógica Computacional  
Dpto. de Ciencias de la Computación e Inteligencia Artificial  
Universidad de Sevilla  
Sevilla, 24 de Septiembre de 2008

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>I Programación funcional</b>	<b>5</b>
1. Programación funcional básica	7
2. Números y funciones	35
3. Estructuras de datos	51
4. Aplicaciones de programación funcional	93
5. Razonamiento sobre programas	103
<b>II Programación lógica</b>	<b>115</b>
6. Introducción a Prolog	117
7. Listas, operadores y aritmética	129
8. Estructuras	139
9. Retroceso, corte y negación	151
10. Programación lógica de segundo orden	161
11. Estilo y eficiencia en programación lógica	175
12. Aplicaciones de PD: Problemas de grafos y de las reinas	193
Bibliografía	201



**Parte I**

**Programación funcional**



# Capítulo 1

## Programación funcional básica

# Programación declarativa (2008–09)

## Tema 1: Programación funcional básica

José A. Alonso Jiménez

## Índice

<b>1. Lenguajes funcionales</b>	<b>1</b>
1.1. De la programación imperativa a la declarativa . . . . .	1
1.2. Historia de la programación funcional . . . . .	2
<b>2. El intérprete de Haskell</b>	<b>3</b>
2.1. Cálculo de expresiones . . . . .	3
2.2. Definición de funciones . . . . .	5
<b>3. Funciones estándar</b>	<b>6</b>
3.1. Constantes, operadores y funciones predefinidas . . . . .	6
3.2. Funciones sobre números . . . . .	7
3.3. Funciones booleanas . . . . .	8
3.4. Funciones sobre listas . . . . .	9
3.5. Funciones sobre funciones . . . . .	10
<b>4. Definición de funciones</b>	<b>10</b>
4.1. Definición por combinación . . . . .	10
4.2. Definición por distinción de casos . . . . .	13
4.3. Definición por análisis de patrones . . . . .	14
4.4. Definición por recursión . . . . .	16
4.5. Indentación y comentarios . . . . .	18
<b>5. Tipado</b>	<b>18</b>
5.1. Clases de errores . . . . .	18
5.2. Expresiones de tipo . . . . .	20
5.3. Polimorfismo . . . . .	22
5.4. Sobrecarga . . . . .	24



# 1. Lenguajes funcionales

## 1.1. De la programación imperativa a la declarativa

### Crisis del software

Problemas:

- ¿Cómo tratar con el tamaño y complejidad de los programas actuales?
- ¿Cómo reducir el tiempo y el coste de desarrollo de los programas?
- ¿Cómo aumentar la confianza en que los programas funcionan correctamente?

Solución de la crisis del software: lenguajes tales que:

- los programas sean claros, concisos y con un alto nivel de abstracción;
- soporten componentes reusables;
- faciliten el uso de verificación formal;
- permitan prototipado rápido y
- tengan herramientas potentes para la resolución de problemas.

### Programación imperativa vs declarativa

- Programación imperativa:
  - Los programas usan **estados implícitos**.
  - Los programas son sucesiones de **órdenes**.
  - Los programas expresan **cómo** se ha de computar.
  - Lenguajes imperativos: Fortran, Pascal, C.
- Programación declarativa:
  - Los programas no usan **estados implícitos**.
  - Los programas son conjunto de **expresiones**.
  - Los programas expresan **qué** se ha de computar.
  - La repetición se realiza mediante **recursión**.

## Tipos de programación declarativa y bases teóricas

Tipos de programación declarativa:

- Programación funcional (o aplicativa):
  - Se basa en el concepto de **función** matemática.
  - Una computación consiste en la aplicación de una función a sus argumentos.
  - Lenguajes funcionales: Haskell, ML, Scheme, Lisp.
- Programación relacional (o lógica):
  - Se basa en el concepto de **relación** matemática.
  - Una computación consiste en la búsqueda de valores de argumentos para satisfacer relaciones.
  - Lenguajes lógicos: Prolog, DLV, Racer.

Bases teóricas:

- Programación imperativa: máquinas de A. Turing (años 30).
- Programación funcional: lambda cálculo de A. Church (años 20).
- Programación lógica: resolución de A. Robinson (años 60).

## 1.2. Historia de la programación funcional

### Historia de la programación funcional

- 1930s: Lambda cálculo (Alonzo Church).
- 1960: Lisp (John McCarthy).
- 1960s: ISWIM (Peter Landin).
- 1970s: FP (John Backus).
- 1970s: ML (Robin Milner).
- 1972: Prolog (Colmerauer).
- 1970s–1980s: Miranda (David Turner).
- 1987: Haskell.
- 2003: Haskell 98 Report.

## 2. El intérprete de Haskell

### 2.1. Cálculo de expresiones

#### Cálculo de expresiones e intérpretes

- Un **programa funcional** es un conjunto de expresiones.
- En los lenguajes funcionales se pueden **definir funciones**.
- Una **computación** es la evaluación de una expresión.
- Las evaluaciones las realizan los **intérpretes**.
- **Hugs** es un intérprete del lenguaje funcional **Haskell**.

#### Inicio, cálculo, ayuda y fin de Hugs

```

PD> hugs
--  --  --  --  ----  --  -----
||  ||  ||  ||  ||  ||  ||__  Hugs 98: Based on the Haskell 98 standard
|_---| |__|| |__|| |__||  _||  Copyright (c) 1994-2005
|_---|      _---|      World Wide Web: http://haskell.org/hugs
||  ||      Bugs: http://hackage.haskell.org/trac/hugs
||  || Version: September 2006 -----

Hugs mode: Restart with command line option +98 for Haskell 98 mode

Type :? for help
Hugs> 2+3
5

Hugs> :?
LIST OF COMMANDS: Any command may be abbreviated to :c where
c is the first character in the full name.

:load <filenames>  load modules from specified files
:load              clear all files except prelude
:also <filenames>  read additional modules
:reload           repeat last load command
:edit <filename>   edit file
:edit             edit last module
:module <module>  set module for evaluating expressions
<expr>           evaluate expression
:type <expr>      print type of expression
:?               display this list of commands
:set <options>    set command line options
:set             help on command line options

```

```
:names [pat]          list names currently in scope
:info <names>         describe named objects
:browse <modules>    browse names exported by <modules>
:main <aruments>     run the main function with the given arguments
:find <name>         edit module containing definition of name
:cd dir              change directory
:gc                 force garbage collection
:version            print Hugs version
:quit              exit Hugs interpreter
Hugs> :q
[Leaving Hugs]
```

### Uso de comandos del sistema operativo

```
Hugs> !pwd
/home/jalonso/alonso/curso-pd
Hugs> !date
mar jul 24 13:01:37 CEST 2007
```

### Ejemplos de evaluaciones

- Evaluación de expresiones aritméticas:

```
Hugs> 5+2*3
11
```

- Evaluación de expresiones matemáticas:

```
Hugs> sqrt(9)
3.0
Hugs> sqrt 9
3.0
Hugs> sin 0.2 * sin 0.2 + cos 0.2 * cos 0.2
1.0
```

- Evaluación de expresiones con listas:

```
Hugs> length [1,3,6]
3
Hugs> reverse [3,4,5,6]
[6,5,4,3]
Hugs> head [1,3,6]
1
```

- Suma de los elementos de una lista:

```
|Hugs> sum [3..6]
|18
```

- Composición de funciones:

```
|Hugs> head (reverse [3..6])
|6
```

## 2.2. Definición de funciones

### Proceso de construcción de un programa

- Llamada al editor:

```
|Hugs> :edit "factorial.hs"
```

- Edición del programa factorial.hs:

```
fact n = product [1..n]
```

- Carga del programa:

```
|Hugs> :load "factorial.hs"
```

- Evaluación:

```
|Main> fact 4
|24
```

### Ampliación del programa

- Números combinatorios:  $\binom{n}{m} = \frac{n!}{k! (n-k)!}$

- Llamada al editor:

```
|Main> :edit "factorial.hs"
```

- Edición:

```
fact n = product [1..n]

comb n k = (fact n) `div` ((fact k) * (fact (n-k)))
```

- Recarga:

```
|Main> :reload "factorial.hs"
```

- Evaluación:

```
|Main> comb 6 2  
|15
```

## 3. Funciones estándar

### 3.1. Constantes, operadores y funciones predefinidas

#### Constantes

- Una **constante** es una función sin parámetros.
- Ejemplo de **definición de una constante**:

```
pi = 3.1416
```

- Ejemplo de uso:

```
|Main> 2*pi  
|6.2832
```

#### Operadores

- Un **operador** es una función con dos parámetros que se escribe entre ellos.
- Ejemplo de **definición de un operador**:

```
n !^! k = (fact n) 'div' ((fact k) * (fact (n-k)))
```

- Ejemplo de uso:

```
|Main> 6 !^! 2  
|15
```

#### Funciones predefinidas y primitivas

- El **preludio** tiene **funciones predefinidas**.
- Consulta de definiciones:

```

Main> :set +E "emacs"
Main> :find fact
-- Abre con emacs el fichero
-- con la definición de factorial.
Main> :find sum
-- Abre con emacs el preludio.
-- Buscando se encuentra la definición
-- sum = foldl' (+) 0

```

- Existen **funciones primitivas** definidas en el intérprete, pero no en Haskell.

## Nombres de funciones y de parámetros

- Ejemplo: en la definición

```
fact n = product [1..n]
```

- el **nombre de la función** es fact
  - el **nombre del parámetro** es n
- Los **nombres de las funciones y de los parámetros** tienen que empezar con una letra minúscula. Después pueden seguir más letras (minúsculas o mayúsculas), pero también números, el símbolo ' y el símbolo \_.
  - Haskell diferencia las mayúsculas de las minúsculas.
  - No se pueden usar como nombres las palabras reservadas.
  - Los **nombres de operadores** son cadenas de símbolos como  
: # % & \* + - = . / \ < > ? ! @ ^ |

## 3.2. Funciones sobre números

### Tipos de números y operadores aritméticos en Haskell

- Tipos de números:
  - Enteros: 17, 0 y -3;
  - Números reales: 2.5, -7.8, 0.0, 1.2e3 y 0.5e-2.
- Operadores aritméticos:
  - Operadores: +, -, \* y /.
  - Ejemplos:

```

5-12    ~> -7
7.2*3.0 ~> 21.6
19/4    ~> 4.75
1.5+2   ~> 3.5

```

- Los operadores aritméticos son primitivos.

### Funciones numéricas predefinidas

- Funciones predefinidas para números enteros:

```

abs      valor absoluto
signum   -1 para negativos, 0 para cero y 1 para positivos
gcd      máximo común divisor
^        potencia

```

- Funciones predefinidas para números reales:

```

sqrt     raíz cuadrada
sin      seno
log      logaritmo natural
exp      exponencial

```

- Funciones predefinidas de transformación entre enteros y reales:

```

FromInteger  de entero a real
round       redondea un real a entero

```

### 3.3. Funciones booleanas

#### Funciones booleanas entre números

Una **función booleana** es una función cuyo valor es un booleano (es decir True o False).

Funciones booleanas entre números:

- Relaciones:

```

<   (menor que)           >   (mayor que),
<=  (menor o igual que)  >=  (mayor o igual que),
==  (igual a)             /=  (distinto de).

```

- Ejemplos:

```

1<2           ~> True
2<1           ~> False
2+3 > 1+4     ~> False
sqrt 2.0 <= 1.5 ~> True
5 /= 1+4      ~> False

```



## Funciones lógicas

- Las **funciones lógicas** son funciones cuyos argumentos y valores son booleanos.

- Funciones lógicas:
  - `&&` (conjunción),
  - `||` (disyunción) y
  - `not` (negación).

- Ejemplos:

```

1<2 && 3<4 ~> True
1<2 && 3>4 ~> False
1<2 || 3>4 ~> True
not False ~> True
not (1<2) ~> False

```

## Otras funciones booleanas predefinidas

- Funciones:
  - `even x` x es par
  - `odd x` x es impar

- Ejemplos:

```

even 7 ~> False
even 6 ~> True
odd (5+2) ~> True

```

## 3.4. Funciones sobre listas

### Funciones predefinidas sobre listas

- `[]` es la lista vacía.
- `x:l` coloca x al frente de l

```

1: [] ~> [1]
1: [3,4,3] ~> [1,3,4,3]

```

- `length l` es el número de elementos de l

```
length [2,4,2] ~> 3
```

- `sum l` es la suma de los elementos de l

```
sum [2,4,2] ~> 8
```

- `l1 ++ l2` es la concatenación de `l1` y `l2`

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

- `null l` se verifica si `l` es la lista vacía

```
| null [] ~> True
| null [1,2] ~> False
```

- `and l` se verifica si todos los elementos de `l` son verdaderos.

```
| and [1<2, 2<3, 1 /= 0] ~> True
| and [1<2, 2<3, 1 == 0] ~> False
```

- `or l` se verifica si algún elemento de `l` es verdadero. Por ejemplo,

```
| or [2>3, 5<9] ~> True
| or [2>3, 9<5] ~> False
```

- `[n..m]` la lista de los números de `n` a `m`

```
| [2..5] ~> [2,3,4,5]
```

- `take n l` es la lista de los `n` primeros elementos de `l`.

```
| take 2 [3,5,4,7] ~> [3,5]
| take 12 [3,5,4,7] ~> [3,5,4,7]
| take 3 [10..20] ~> [10,11,12]
```

### 3.5. Funciones sobre funciones

#### Funciones con funciones como parámetros

- `map f l` es la lista obtenida aplicando `f` a cada elemento de `l`

```
| map fact [1,2,3,4,5] ~> [1,2,6,24,120]
| map sqrt [1,2,4] ~> [1.0,1.4142135623731,2.0]
| map even [1..5] ~> [False,True,False,True,False]
```

## 4. Definición de funciones

### 4.1. Definición por combinación

#### Definiciones por combinación con un parámetro

- `fact n` es el factorial de `n`. Por ejemplo,

```
| fact 4 ~> 24
```

```
fact n = product [1..n]
```

- `impar x` se verifica si el número `x` es impar. Por ejemplo,

```
impar 7 ~> True
impar 6 ~> False
```

```
impar x = not (even x)
```

- `cuadrado x` es el cuadrado del número `x`. Por ejemplo,

```
| cuadrado 3 ~> 9
```

```
cuadrado x = x*x
```

- `suma_de_cuadrados l` es la suma de los cuadrados de los elementos de la lista `l`. Por ejemplo,

```
| suma_de_cuadrados [1,2,3] ~> 14
```

```
suma_de_cuadrados l = sum (map cuadrado l)
```

### Definiciones por combinación con varios parámetros

- `n_combinaciones n k` es el número de maneras de escoger `k` elementos de un conjunto de `n` elementos. Por ejemplo,

```
| n_combinaciones 6 2 ~> 15
```

```
n_combinaciones n k =
  (fact n) 'div' ((fact k) * (fact (n-k)))
```

- `raices a b c` es la lista de las raíces de la ecuación  $ax^2 + bx + c = 0$ . Por ejemplo,

```
| raices 1 3 2 ~> [-1.0,-2.0]
```

```
raices a b c = [ (-b+sqrt(b*b-4*a*c))/(2*a),
                 (-b-sqrt(b*b-4*a*c))/(2*a) ]
```

**Definiciones por combinación sin parámetros**

- iva el porcentaje de IVA.

```
iva = 0.16
```

- cambioEuro es el cambio de un euro a pesetas

```
cambioEuro = 166.386
```

**Forma de las definiciones de las funciones**

- el nombre de la función
- los nombres de los parámetros (si existen)
- el símbolo =
- una expresión, que puede contener los parámetros, las funciones estándar y otras funciones definidas.

**Definiciones por combinación de funciones booleanas**

- Ejemplos:

```
negativo x = x < 0  
positivo x = x > 0  
esCero   x = x == 0
```

- Notar la diferencia entre = y == en la definición de esCero.

**Definiciones con entornos locales**

- Ejemplo de redefinición de raices

```
raices' a b c = [ (-b+d)/n, (-b-d)/n ]  
  where d = sqrt(b*b-4*a*c)  
        n = 2*a
```

- Mejora en legibilidad y en eficiencia.

```

Main> :set +s
Main> raices 1 3 2
[-1.0,-2.0]
(134 reductions, 242 cells)
Main> raices' 1 3 2
[-1.0,-2.0]
(104 reductions, 183 cells)

```

## 4.2. Definición por distinción de casos

### Definiciones por distinción de casos

- `abs x` es el valor absoluto de `x`. Por ejemplo,

```
abs (-3) ~> 3
```

Prelude
<pre>abs x   x &lt; 0 = -x         x &gt;= 0 = x</pre>

- `signum x` es 1 si `x` es positivo, 0 si `x` es cero y -1 si `x` es negativo. Por ejemplo,

```
signum 7 ~> 1
signum (-4) ~> -1
```

Prelude
<pre>signum x   x &gt; 0 = 1             x == 0 = 0             otherwise = -1</pre>

- Las definiciones de los diferentes casos son precedidas por expresiones booleanas, que se llaman **guardas**.
- En cada **expresión guardada** debe existir el símbolo `|`, una expresión booleana, el símbolo `=` y una expresión.
- Forma de las definiciones de las funciones:
  - el nombre de la función
  - los nombres de los parámetros (si existen)
  - el símbolo `=` y una expresión, o una o más expresiones guardadas.
  - si se desea, la palabra `where` seguida de definiciones locales.

### 4.3. Definición por análisis de patrones

#### Clases de parámetros

- **Parámetros formales:** los usados en las definiciones de funciones.
- **Parámetros actuales:** los usados en las llamadas de funciones.
- Ejemplos:

- en la definición

```
f x y = x * y
```

los parámetros formales son  $x$  e  $y$ .

- en la expresión

```
| f 7 (2+3)
```

los parámetros actuales son 7 y  $2+3$ .

- En la evaluación de una expresión se sustituyen los parámetros formales por los parámetros actuales.
- Los parámetros formales, hasta ahora, son nombres.
- Los parámetros actuales son expresiones.

#### Definiciones con patrones

- $g\ 1$  es la suma de los dos elementos de  $1$  si  $1$  es una lista de 3 números el primero de los cuales es el número 1.

```
g' [1,x,y] = x+y
```

Por ejemplo,

```
Main> g [1,3,5]
8
Main> g [2,3,5]
Program error: pattern match failure: g [2,3,5]
Main> g [1,3,5,4]
Program error: pattern match failure: g [1,3,5,4]
```

- $suma\ 1$  es la suma de los elementos de la lista  $1$  cuando  $1$  es una lista con tres elementos como máximo.

```

suma []      = 0
suma [x]     = x
suma [x,y]   = x+y
suma [x,y,z] = x+y+z

```

Por ejemplo,

```

Main> suma [1,3]
4
Main> suma [1,3,4,5]
Program error: pattern match failure: suma [1,3,4,5]

```

### Construcciones que se pueden utilizar como patrón

- Números (por ejemplo 3);
- Las constantes True y False;
- Nombres (por ejemplo, x);
- Listas cuyos elementos también son patrones (por ejemplo [1, x, y]);
- El operador : con patrones a la izquierda y a la derecha (por ejemplo a:b);
- El operador + con un patrón a la izquierda y un entero a la derecha (por ejemplo n+1);

### Ejemplos de emparejamiento de patrones

Patrón	Argumento	Sustitución
x	0	{x/0}
0	0	{}
(x:y)	[1,2]	{x/1, y/[2]}
(1:x)	[1,2]	{x/[2]}
(1:x)	[2,3]	Fallo
(x:_:_:y)	[1,2,3,4,5,6]	{x/1, y/[4,5,6]}
[]	[]	{}
[1,x]	[1,2]	{x/2}
[x,y]	[1]	Fallo
(n+2)	6	{n/4}
(n+1)	0	Fallo

## Definiciones con patrones en el prelude

- `x && y` es la conjunción de `x` e `y`

```

Prelude
False && x  = False
True  && x  = x

```

- `head l` es la cabeza de la lista `l`. Por ejemplo,

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

```

Prelude
head (x:_) = x

```

- `tail l` es el resto de la lista `l`. Por ejemplo,

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

```

Prelude
tail (_:xs) = xs

```

- Patrón de [variable anónima](#): `_`

## Definición con patrones aritméticos

- `anterior x` es el anterior de `x`. Por ejemplo,

```

Main> anterior 3
2
Main> anterior 0
Program error: pattern match failure: anterior 0

```

```
anterior (n+1) = n
```

## 4.4. Definición por recursión

### Ejemplos de definiciones por recursión numérica

- `fac n` es el factorial del número `n`. Por ejemplo,

```
| fac 20 ~> 2432902008176640000
```



```
fac 0      = 1
fac (n+1) = (n+1) * fac n
```

- potencia  $x$  y es  $x^y$ . Por ejemplo,

```
| potencia 2 4 ~> 16
```

```
potencia x 0      = 1
potencia x (n+1) = x * (potencia x n)
```

### Cálculo paso a paso del factorial

Cálculo paso a paso de fac 4:

```
fac 0      = 1          -- fac.1
fac (n+1) = (n+1) * fac n -- fac.2
```

```
fac 4
= 4 * fac 3          [por fac.2]
= 4 * (3 * fac 2)   [por fac.2]
= 4 * (3 * (2 * fac 1)) [por fac.2]
= 4 * (3 * (2 * (1 * fac 0))) [por fac.2]
= 4 * (3 * (2 * (1 * 1))) [por fac.1]
= 4 * (3 * (2 * 1))     [por aritmética]
= 4 * (3 * 2)          [por aritmética]
= 4 * 6                [por aritmética]
= 24                   [por aritmética]
```

### Ejemplos de definiciones por recursión sobre listas

- `sum l` es la suma de los elementos de  $l$ . Por ejemplo,

```
| sum [1,3,6] ~> 10
```

----- Prelude -----

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

- `longitud l` es el número de elementos de  $l$ . Por ejemplo,

```
| longitud [1,3,6] ~> 3
```

```
longitud []      = 0
longitud (_,xs) = 1 + longitud xs
```

### Cálculo paso a paso de la longitud

Cálculo paso a paso de longitud [3,2,3,5]:

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

```
longitud [3,2,3,5]
= 1 + longitud [2,3,5]           [por longitud.2]
= 1 + (1 + longitud [3,5])       [por longitud.2]
= 1 + (1 + (1 + longitud [5]))   [por longitud.2]
= 1 + (1 + (1 + (1 + longitud []))) [por longitud.2]
= 1 + (1 + (1 + (1 + 0)))        [por longitud.1]
= 1 + (1 + (1 + 1))              [por aritmética]
= 1 + (1 + 2)                    [por aritmética]
= 1 + 3                           [por aritmética]
= 4                               [por aritmética]
```

## 4.5. Indentación y comentarios

### Indentación y comentarios

- Haskell usa una sintaxis bidimensional.
- Tipos de comentarios:
  - hasta fin de línea: --
  - región comentada: {- ... -}

## 5. Tipado

### 5.1. Clases de errores

#### Errores sintácticos

- Ejemplo de **error sintáctico**: Sea ej1.hs un fichero con el siguiente contenido

Código erróneo

```
esCero x = x=0
```

al cargarlo, se produce el siguiente error:

```
Hugs> :l ej1.hs
ERROR "ej1.hs":1 - Syntax error in input (unexpected '=')
```

## Errores de dependencia

- Ejemplo de **error de dependencia** (i.e. uso de funciones no definidas): Sea `ej2.hs` un fichero con el siguiente contenido

```
_____ Código erróneo _____  
fact n = producto [1..n]
```

al cargarlo, se produce el siguiente error:

```
Hugs> :l ej2.hs  
ERROR "ej2.hs":1 - Undefined variable "producto"
```

## Errores de tipo

- Ejemplo de **error de tipo**: Sea `ej3.hs` un fichero con el siguiente contenido

```
_____ Código erróneo _____  
f x = 2*x + True
```

al cargarlo, se produce el siguiente error:

```
Hugs> :l "ej3.hs"  
ERROR "ej3.hs":1 - Instance of Num Char required for definition of ff
```

- Ejemplo de **error de tipo** en evaluación:

```
Hugs> length 3  
ERROR - Unresolved overloading  
*** Type      : Num [a] => Int  
*** Expression : length 3
```

## Fases de análisis

1. análisis sintáctico
2. análisis de dependencias
3. análisis de tipo

## 5.2. Expresiones de tipo

### Determinación del tipo de una expresión

- Mediante `:type`

```
Hugs> :type 2+3
2 + 3 :: Num a => a
```

- Mediante la activación de escritura de tipos:

```
Hugs> :set +t
Hugs> 2+3
5 :: Integer
```

que se lee “5 es de tipo Integer”.

### Tipos básicos

- Int**: el tipo de los enteros en el intervalo  $[-2^{29}, 2^{29} - 1]$ .
- Integer**: el tipo de los enteros de precisión ilimitada.
- Float**: el tipo de los números reales.
- Double**: el tipo de los reales con mayor precisión que `Float`.
- Bool**: el tipo de los valores booleanos: `True` y `False`.

Nota: Los nombres de los tipos comienzan por mayúscula.

### Tipo lista

- Ejemplos de tipo lista:

```
Hugs> :set +t
Hugs> [1,2,3]
[1,2,3] :: [Integer]
Hugs> [True,False]
[True,False] :: [Bool]
Hugs> [[1,2],[2,3,4]]
[[1,2],[2,3,4]] :: [[Integer]]
```

- `[a]` es el tipo de las listas de elementos del tipo `a`.
- Todos los elementos de una lista tienen que ser del mismo tipo:

```
Hugs> [1,True]
ERROR - Unresolved overloading
*** Type      : Num Bool => [Bool]
*** Expression : [1,True]
```

## Tipo de funciones

- Determinación del tipo de una función:

```
Hugs> :type length
length :: [a] -> Int
Hugs> :type sqrt
sqrt :: Floating a => a -> a
Hugs> :type even
even :: Integral a => a -> Bool
Hugs> :type sum
sum :: Num a => [a] -> a
```

- Determinación del tipo de una lista de funciones:

```
Hugs> :type [sin, cos, tan]
[sin,cos,tan] :: Floating a => [a -> a]
```

## Inferencia de tipos

El intérprete puede determinar automáticamente los tipos. Por ejemplo, si se define

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

entonces se puede determinar el tipo de f

```
Main> :type f
f :: Num a => [a] -> a
```

## Declaración de tipos (Signaturas)

- Se puede escribir el tipo de una función en el programa. Por ejemplo,

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

entonces se puede comprobar el tipo

```
|Main> :type g
|g :: [Int] -> Int
```

- Ventajas de la declaración de tipos:
  - se comprueba si la función tiene el tipo que está declarado
  - la declaración del tipo ayuda a entender la función.
- No hace falta escribir la declaración directamente delante la definición.

### 5.3. Polimorfismo

#### Ejemplo de función polimorfa

- Ejemplo:

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

Se lee “length aplica una lista de elementos de un tipo a en un número entero de precisión limitada”.

- Notas:
  - a es una [variable de tipo](#).
  - Un tipo que contiene variables de tipo, se llama [tipo polimórfico](#).
  - Las funciones con un tipo polimórfico se llaman [funciones polimórficas](#).
  - El fenómeno en sí se llama [polimorfismo](#).

#### Otros ejemplos de funciones polimorfas

- Ejemplo de polimorfismo con listas:

```
|Main> :type head
|head :: [a] -> a
```

- Ejemplo de polimorfismo sin listas:

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

Por ejemplo,

```

Main> :type id True
id True :: Bool
Main> :type id [True, False]
id [True,False] :: [Bool]
Main> :type id 3
id 3 :: Num a => a

```

### Ejemplo de tipos de funciones de más de un parámetro

- `map f l` es la lista obtenida aplicando `f` a cada elemento de `l`

```

map fact [1,2,3,4,5] ~> [1,2,6,24,120]
map sqrt [1,2,4] ~> [1.0,1.4142135623731,2.0]
map even [1..5] ~> [False,True,False,True,False]

```

- Tipo de `map`:

```

Hugs> :type map
map :: (a -> b) -> [a] -> [b]

```

### Ejemplo de tipo de operadores

- `l1 ++ l2` es la concatenación de `l1` y `l2`

```

[2,3] ++ [3,2,4,1] ~> [2,3,3,2,4,1]

```

- Tipo de `(++)`

```

Main> :type (++)
(++): :: [a] -> [a] -> [a]

```

- `x && y` es la conjunción de `x` e `y`. Por ejemplo,

```

1<2 && 3<4 ~> True
1<2 && 3>4 ~> False

```

- Tipo de `(&&)`

```

Main> :type (&&)
(&&) :: Bool -> Bool -> Bool

```

## 5.4. Sobrecarga

### Diferencia entre polimorfismo y sobrecarga

- Ejemplo de función polimorfa:

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

- Ejemplo de operador sobrecargado:

```
|Main> :type (+)
|(+ :: Num a => a -> a -> a
```

se lee “+ es de tipo  $a \rightarrow a \rightarrow a$  donde  $a$  tiene tipo en la clase `Num`”.

### Clases

Una **clase** es un grupo de tipos con una característica en común.

Clases predefinidas en el preludio:

- **Num** es la clase de tipos cuyos elementos se pueden sumar, multiplicar, restar y dividir (tipos numéricos);
- **Ord** es la clase de tipos cuyos elementos se pueden ordenar;
- **Eq** es la clase cuyos elementos se pueden comparar (en inglés: “equality types”).

### Ejemplos de funciones sobrecargadas

- Otros ejemplos de operadores sobrecargados:

```
|Main> :type (<)
|(<) :: Ord a => a -> a -> Bool
|Main> :type (==)
|(<=) :: Eq a => a -> a -> Bool
```

- Ejemplo de función definida sobrecargada:

```
cubo x = x * x * x
```

cuyo tipo es

```
|Main> :type cubo
|cubo :: Num a => a -> a
```



---

## Bibliografía

1. H. C. Cunningham (2007) *Notes on Functional Programming with Haskell*.
2. J. Fokker (1996) *Programación funcional*.
3. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J. Gallardo (2004). *Razonando con Haskell (Un curso sobre programación funcional)*.
4. S. Thompson (1999) *Haskell: The Craft of Functional Programming*.
5. E.P. Wentworth (1994) *Introduction to Funcional Programming*.



## **Capítulo 2**

# **Números y funciones**

# Programación declarativa (2008–09)

Tema 2: Números y funciones

José A. Alonso Jiménez

## Índice

<b>1. Operadores</b>	<b>1</b>
1.1. Operadores como funciones y viceversa . . . . .	1
1.2. Precedencias . . . . .	1
1.3. Asociatividad . . . . .	2
1.4. Definición de operadores . . . . .	2
<b>2. Currificación</b>	<b>3</b>
2.1. Instanciación parcial . . . . .	3
2.2. Reducción de paréntesis . . . . .	4
2.3. Secciones de operadores . . . . .	5
<b>3. Funciones como parámetros</b>	<b>6</b>
3.1. Funciones sobre listas . . . . .	6
3.2. Iteración . . . . .	6
3.3. Composición . . . . .	7
3.4. Intercambio de argumentos . . . . .	7
3.5. Funciones anónimas . . . . .	7
<b>4. Problemas de cálculo numérico</b>	<b>8</b>
4.1. Cálculo con números enteros . . . . .	8
4.2. Diferenciación numérica . . . . .	11
4.3. Cálculo de la raíz cuadrada . . . . .	12
4.4. Ceros de una función . . . . .	12
4.5. Funciones inversas . . . . .	13

# 1. Operadores

## 1.1. Operadores como funciones y viceversa

### Operadores como funciones

- Ejemplo:

2+3	↪	5
(+)	2 3	↪ 5

- Regla: un operador entre paréntesis se comporta como la función correspondiente.

### Funciones como operadores

- Ejemplo:

max	2 3	↪	3
2	'max'	3	↪ 3

- Regla: una función entre comillas inversas se comporta como el operador correspondiente.

## 1.2. Precedencias

### Ejemplos de precedencias

- Ejemplos:

2*3+4*5	↪	26
2*3<7	↪	True

\* tiene mayor precedencia que +  
 < tiene mayor precedencia que \*

- Pueden usarse paréntesis para saltarse las precedencia. Por ejemplo,

2*3+4*5	↪	26
2*(3+4)*5	↪	70

**Precedencia de los operadores del prelude**

nivel 9	., !!
nivel 8	^
nivel 7	*, /, 'quot', 'rem', 'div', 'mod'
nivel 6	+, -
nivel 5	:, ++
nivel 4	==, /=, <, <=, >=, >, 'elem', 'notElem'
nivel 3	&&
nivel 2	

**Precedencia de la llamada a funciones**

- La llamada a funciones (el operador invisible entre  $f$  y  $x$  en  $f\ x$ ) tiene la máxima precedencia. Por ejemplo,

cuadrado 3 + 4	$\rightsquigarrow$	13
cuadrado (3 + 4)	$\rightsquigarrow$	49

- En la definición de funciones por análisis de patrones la llamada a función tiene la precedencia más alta. Por ejemplo,

fac 0	=	1
fac (n+1)	=	(n+1) * fac n

los paréntesis en la segunda ecuación son necesarios.

**1.3. Asociatividad****Operadores**

- asocian por la izquierda:  
+, -, \*, /, 'quot', 'rem', 'div', 'mod', &&, ||, !!
- asocian por la derecha:  
^, :, ++, .
- no asocian:  
==, /=, <, <=, >=, >, 'elem', 'notElem'

**1.4. Definición de operadores****Definición de operadores**

- Ejemplo de declaración de operadores del prelude:

Prelude

```
infixr 8  ^
infixl 6  +, -
infix 4   ==, /=, <, <=, >=, >, 'elem', 'notElem'
```

- Ejemplo de definición de operador:  
 $x \sim= y$  se verifica si  $|x - y| < 0,0001$ . Por ejemplo,

```
3.00001 ~= 3.00002 ~> True
3.1 ~= 3.2      ~> False
```

```
infix 4 ~=
(~=)    :: Float -> Float -> Bool
x ~= y  = abs(x-y) < 0.0001
```

## 2. Currificación

### 2.1. Instanciación parcial

#### Ejemplo de instanciación parcial

Definición de siguiente mediante instanciación parcial de suma:

- `suma x y` es la suma de `x` e `y`:

```
suma :: Int -> Int -> Int
suma x y = x+y
```

- `siguiente x` es el siguiente de `x`. Por ejemplo,

```
siguiente 3 ~> 4
```

```
siguiente :: Int -> Int
siguiente = suma 1
```

#### Instancias parciales

- Ejemplo de uso de una instancia parcial como parámetro:

```
map (suma 5) [1,2,3] ~> [6,7,8]
```

- Tipos de instancias parciales:

```

Main> :type suma
suma :: Int -> Int -> Int
Main> :type suma 1
suma 1 :: Int -> Int
Main> :type suma 1 3
suma 1 3 :: Int

```

Regla:  $\rightarrow$  asocia por la derecha.

- Estrategia de currificación: describir las funciones de más de un parámetro por funciones de un parámetro que devuelven otra función.

## 2.2. Reducción de paréntesis

### Ejemplos de reducción de paréntesis

- Se ha elegido la asociatividad de  $\rightarrow$  y la aplicación de función de tal manera que no se ve la currificación: la aplicación de función asocia por la izquierda y  $\rightarrow$  asocia por la derecha. Por ejemplo,

```

sumaCuatro :: Int -> Int -> Int -> Int -> Int
sumaCuatro a b c d = a+b+c+d

```

entonces

```

sumaCuatro 2 3 4 5 ~> 14

```

- La regla es: si no están escritos los paréntesis, entonces están escritos implícitamente de tal manera que funciona la currificación.

### Ejemplos en los que se necesitan paréntesis

- en declaraciones de tipo:

```

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

```

- en expresiones:

```

cuadrado (3 + 4)

```



## 2.3. Secciones de operadores

### Secciones de operadores

Si  $\oplus$  es un operador, entonces

- $(\oplus x)$  es la función definida por  $(\oplus x)y = y \oplus x$ .
- $(x \oplus)$  es la función definida por  $(x \oplus)y = x \oplus y$ .

Ejemplos de definiciones mediante secciones:

- `succ x` es el siguiente de `x`. Por ejemplo, `succ 3`  $\rightsquigarrow$  4

```
----- Prelude -----
succ = (+1)
```

- `doble x` es el doble de `x`. Por ejemplo, `doble 3`  $\rightsquigarrow$  6

```
doble = (2*)
```

- `mitad x` es la mitad de `x`. Por ejemplo, `mitad 5`  $\rightsquigarrow$  2.5

```
mitad = (/2)
```

- `inverso x` es el inverso de `x`. Por ejemplo, `inverso 2`  $\rightsquigarrow$  0.5

```
inverso = (1/)
```

- `cuadrado x` es el cuadrado de `x`. Por ejemplo, `cuadrado 3`  $\rightsquigarrow$  9

```
cuadrado = (^2)
```

- `dosElevadoA x` es  $2^x$ . Por ejemplo, `dosElevadoA 3`  $\rightsquigarrow$  8

```
dosElevadoA = (2^)
```

- `esPositivo` se verifica si `x` es positivo. Por ejemplo,

```
esPositivo 3     $\rightsquigarrow$  True
esPositivo (-3)  $\rightsquigarrow$  False
```

```
esPositivo = (>0)
```

### Secciones como parámetros

Ejemplos de uso de secciones como parámetros:

```
Main> map (2*) [1,2,3]
[2,4,6]
Main> map (2/) [1,2,4,8]
[2.0,1.0,0.5,0.25]
Main> map (/2) [1,2,4,8]
[0.5,1.0,2.0,4.0]
Main> map (2^) [1,2,4,8]
[2,4,16,256]
Main> map (^2) [1,2,4,8]
[1,4,16,64]
```

## 3. Funciones como parámetros

### 3.1. Funciones sobre listas

#### Funciones de orden superior

Las [funciones de orden superior](#) son las que tienen funciones como parámetros.

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

```
|map (2*) [1,2,3] ~> [2,4,6]
```

- `filter p l` es la lista de los elementos de `l` 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]
```

### 3.2. Iteración

#### La función de iteración `until`

- `until p f x` aplica la `f` a `x` el menor número posible de veces, hasta alcanzar un valor que satisface el predicado `p`. Por ejemplo,

```
|until (>1000) (2*) 1 ~> 1024
```

```
----- Prelude -----
until :: (a -> Bool) -> (a -> a) -> a -> a
until p f x = if p x
              then x
              else until p f (f x)
```

### 3.3. Composición

#### Composición de funciones

- `f . g` es la composición de las funciones `f` y `g`; es decir, la función que aplica `x` en `f(g(x))`. Por ejemplo,

```
(cuadrado . succ) 2 ~> 9
(succ . cuadrado) 2 ~> 5
```

Prelude

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

- Ejemplo de definición mediante composición:

```
impar = not . even
```

- Ejemplo de uso de composición como parámetro:

```
Main> filter (not . even) [1,3,5,4,2,6,1]
[1,3,5,1]
```

### 3.4. Intercambio de argumentos

#### Intercambio de argumentos

- `flip f` intercambia el orden de los argumentos de `f`. Por ejemplo,

```
(-) 5 2 ~> 3
flip (-) 5 2 ~> -3
(/) 5 2 ~> 2.5
flip (/) 5 2 ~> 0.4
```

Prelude

```
flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x
```

### 3.5. Funciones anónimas

#### Funciones anónimas

La expresión

```
\x1 ... xn -> cuerpo
```

representa una función anónima. Por ejemplo,

```
Main> map (\x -> 2*x+1) [1..5]
[3,5,7,9,11]
Main> filter (\x -> 2*x > 7) [1..5]
[4,5]
Main> filter (\x -> odd x) [1..5]
[1,3,5]
Main> filter (\x -> 2*x > 7 && odd x) [1..5]
[5]
```

## 4. Problemas de cálculo numérico

### 4.1. Cálculo con números enteros

#### Cociente y resto

- `div x y` es el cociente entero de  $x$  entre  $y$ . Por ejemplo,

```
|7 'div' 2 ~> 3
```

- `rem x y` es el resto de dividir  $x$  por  $y$ . Por ejemplo,

```
|7 'rem' 2 ~> 1
```

- Aplicaciones de las funciones cociente y resto:

- Calcular horas: Si son las 9 ahora, entonces 33 horas más tarde serán las  $(9+33) \text{ 'rem' } 24 = 20$ .
- Calcular días: Si se codifican los días con 0=domingo, 1=lunes, ..., 6=sábado y hoy es el día 3 (miércoles), entonces dentro de 40 días será  $(3+40) \text{ 'rem' } 7 = 1$  (lunes).
- Determinar si un número es divisible: Un número es divisible por  $n$  si el resto de la división por  $n$  es igual a cero.
- Determinar las cifras de un número:
  - La última cifra de un número  $x$  es  $x \text{ 'rem' } 10$ .
  - La penúltima cifra es  $(x \text{ 'div' } 10) \text{ 'rem' } 10$ .
  - La antepenúltima cifra es  $(x \text{ 'div' } 100) \text{ 'rem' } 10$ , etc.

#### Diseño ascendente: Cálculo de una lista de primos

- `divisible x y` se verifica si  $x$  es divisible por  $y$ . Por ejemplo,

```
|divisible 9 3 ~> True
|divisible 9 2 ~> False
```

```
divisible :: Int -> Int -> Bool
divisible x y = x `rem` y == 0
```

- `divisores x` es la lista de los divisores de `x`. Por ejemplo,

```
|divisores 12 ~> [1,2,3,4,6,12]
```

```
divisores :: Int -> [Int]
divisores x = filter (divisible x) [1..x]
```

- `primo x` se verifica si `x` es primo. Por ejemplo,

```
|primo 5 ~> True
|primo 6 ~> False
```

```
primo :: Int -> Bool
primo x = divisores x == [1,x]
```

- `primos x` es la lista de los números primos menores o iguales que `x`. Por ejemplo,

```
|primos 40 ~> [2,3,5,7,11,13,17,19,23,29,31,37]
```

```
primos :: Int -> [Int]
primos x = filter primo [1..x]
```

### Diseño descendente: Cálculo del día de la semana

- `día d m a` es el día de la semana correspondiente al día `d` del mes `m` del año `a`. Por ejemplo,

```
|día 31 12 2007 ~> "lunes"
```

```
día d m a = díaSemana ((númeroDeDías d m a) `mod` 7)
```

- `númeroDía d m a` es el número de días transcurridos desde el 1 de enero del año 0 hasta el día `d` del mes `m` del año `a`. Por ejemplo,

```
|númeroDeDías 31 12 2007 ~> 733041
```

```
númeroDeDías d m a = (a-1)*365
                    + númeroDeBisiestos a
                    + sum (take (m-1) (meses a))
                    + d
```

- `númeroDeBisiestos a` es el número de años bisiestos antes del año `a`.

```
númeroDeBisiestos a =
  length (filter bisiesto [1..a-1])
```

- `bisiesto a` se verifica si el año `a` es bisiesto. La definición de año bisiesto es
  - un año divisible por 4 es un año bisiesto (por ejemplo 1972);
  - excepción: si es divisible por 100, entonces no es un año bisiesto
  - excepción de la excepción: si es divisible por 400, entonces es un año bisiesto (por ejemplo 2000).

```
bisiesto a =
  divisible a 4
  && (not(divisible a 100) || divisible a 400)
```

- `meses a` es la lista con el número de días de los meses del año `a`. Por ejemplo,

```
| meses 2000 ~> [31,29,31,30,31,30,31,31,30,31,30,31]
```

```
meses a = [31,feb,31,30,31,30,31,31,30,31,30,31]
  where feb | bisiesto a = 29
           | otherwise = 28
```

- `take n l` es la lista de los `n` primeros elementos de `l`. Por ejemplo,

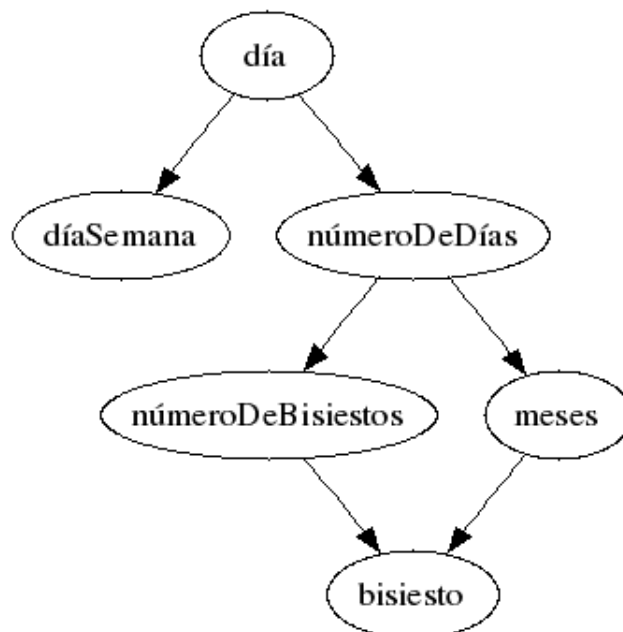
```
| take 2 [3,5,4,7] ~> [3,5]
| take 12 [3,5,4,7] ~> [3,5,4,7]
```

- `díaSemana n` es el `n`-ésimo día de la semana comenzando con 0 el domingo. Por ejemplo,

```
| díaSemana 2 ~> "martes"
```

```
díaSemana 0 = "domingo"
díaSemana 1 = "lunes"
díaSemana 2 = "martes"
díaSemana 3 = "miércoles"
díaSemana 4 = "jueves"
díaSemana 5 = "viernes"
díaSemana 6 = "sábado"
```

- Grafo de funciones



## 4.2. Diferenciación numérica

### Diferenciación numérica

- derivada a f x es el valor de la derivada de la función f en el punto x con aproximación a. Por ejemplo,

```
derivada 0.001 sin pi ~ -0.9999273
derivada 0.001 cos pi ~ 0.0004768371
```

```
derivada :: Float -> (Float -> Float) -> Float -> Float
derivada a f x = (f(x+a)-f(x))/a
```

- Tipos de derivadas:

```
derivadaBurda = derivada 0.01
derivadaFina  = derivada 0.0001
derivadaSuper = derivada 0.000001
```

Por ejemplo,

```
|derivadaFina cos pi ~> 0.0
```

- `derivadaFinaDelSeno x` es el valor de la derivada fina del seno en  $x$ . Por ejemplo,

```
|derivadaFinaDelSeno pi ~> -0.9989738
```

```
derivadaFinaDelSeno = derivadaFina sin
```

### 4.3. Cálculo de la raíz cuadrada

#### Cálculo iterativo de la raíz cuadrada

- Propiedad: si  $y$  es una aproximación de  $\sqrt{x}$ , entonces  $\frac{1}{2}(y + \frac{x}{y})$  es una aproximación mejor.
- Propiedad:  $\sqrt{x}$  es el límite de la sucesión  $x_n$  definida por
 
$$\begin{cases} x_0 &= 1 \\ x_{n+1} &= \frac{1}{2}(y + \frac{x}{x_n}) \end{cases}$$
- Programa: `raiz x` es la raíz cuadrada de  $x$  calculada usando la propiedad anterior. Por ejemplo,

```
|raiz 9 ~> 3.00000000139698
```

```
raiz x = until acceptable mejorar 1
      where mejorar y = 0.5*(y+x/y)
            acceptable y = abs(y*y-x) < 0.00001
```

### 4.4. Ceros de una función

#### Método de Newton para calcular los ceros

- Propiedad: si  $b$  es una aproximación para el punto cero de  $f$ , entonces  $b - \frac{f(b)}{f'(b)}$  es una mejor aproximación.



- Propiedad: el límite de la sucesión  $x_n$  definida por

$$\begin{cases} x_0 &= 1 \\ x_{n+1} &= x_n - \frac{f(x_n)}{f'(x_n)} \end{cases}$$

es un cero de  $f$ .

- Programa: `puntoCero f` es un cero de la función  $f$  calculado usando la propiedad anterior. Por ejemplo,

```
| puntoCero cos ~> 1.570796
```

```
puntoCero f = until acceptable mejorar 1
  where mejorar b = b - f b / derivadaFina f b
        acceptable b = abs (f b) < 0.00001
```

## 4.5. Funciones inversas

### Funciones inversas

- Definición de raíces mediante `puntoCero`:

```
raíz_cuadrada a = puntoCero f
  where f x = x*x-a

raíz_cúbica a = puntoCero f
  where f x = x*x*x-a
```

- Definición de funciones inversas mediante `puntoCero`:

```
arco_seno a = puntoCero f
  where f x = sin x - a

arco_coseno a = puntoCero f
  where f x = cos x - a
```

- Patrón de función inversa:

```
inversa g a = puntoCero f
  where f x = g x - a
```

- Redefiniciones con el patrón:

```
arco_seno_1    = inversa sin
arco_coseno_1  = inversa cos
logaritmo      = inversa exp
```

Por ejemplo,

```
|logaritmo (exp 1) ~> 1.0
```

## Bibliografía

1. H. C. Cunningham (2007) *Notes on Functional Programming with Haskell*.
2. J. Fokker (1996) *Programación funcional*.
3. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J. Gallardo (2004). *Razonando con Haskell (Un curso sobre programación funcional)*.
4. S. Thompson (1999) *Haskell: The Craft of Functional Programming*.
5. E.P. Wentworth (1994) *Introduction to Functional Programming*.

# Capítulo 3

## Estructuras de datos

# Programación declarativa (2008–09)

Tema 3: Estructuras de datos

José A. Alonso Jiménez

## Índice

<b>1. Listas</b>	<b>1</b>
1.1. Construcción de listas . . . . .	1
1.2. Funciones sobre listas . . . . .	3
1.3. Funciones de orden superior sobre listas . . . . .	10
1.4. Ordenación de listas . . . . .	14
<b>2. Listas especiales</b>	<b>17</b>
2.1. Cadenas . . . . .	17
2.2. Caracteres . . . . .	17
2.3. Funciones de cadenas y caracteres . . . . .	17
2.4. Listas infinitas y evaluación perezosa . . . . .	19
<b>3. Tuplas</b>	<b>25</b>
3.1. Uso de tuplas . . . . .	25
3.2. Listas y tuplas . . . . .	28
3.3. Tuplas y currificación . . . . .	30
<b>4. Tipos de datos definidos</b>	<b>30</b>
4.1. Definiciones de tipos . . . . .	30
4.2. Números racionales . . . . .	32
4.3. Tipo de dato definido: Árboles . . . . .	33
4.4. Árboles de búsqueda . . . . .	35
4.5. Usos especiales de definiciones de datos . . . . .	37

# 1. Listas

## 1.1. Construcción de listas

### Listas

- Las **listas** se usan para agrupar varios elementos.
- Los **elementos** de una lista tienen que ser del mismo tipo.
- El **tipo de una lista** se indica escribiendo el tipo de sus elementos entre corchetes.
- Maneras de **construir** listas:
  - enumeración
  - con el operador (`:`)
  - intervalos numéricos
  - por comprensión

### Construcción de listas por enumeración

- Ejemplos de listas por enumeración:

```
[1,2,3]           :: [Int]
[True,False,True] :: [Bool]
[sin, cos, tan]   :: [Float -> Float]
[[1,2,3],[1,4]]   :: [[Int]]
```

- Ejemplos de listas con expresiones:

```
[1+2, 3*4, length [1,2,4,5]] :: [Int]
[3<4, 2+3==4+1, (2<3) && (2>3)] :: [Bool]
[map (1+), filter (2<)]       :: [[Int] -> [Int]]
```

- Ejemplos de listas unitarias:

```
[3] :: [Int]
[[True, 3<2]] :: [[Bool]]
```

La lista vacía es polimorfa:

```
:set +t
tail [3]    ~> [] :: [Integer]
tail [True] ~> [] :: [Bool]

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

```

length []    ~> 0 :: Int
:t sum      ~> sum :: Num a => [a] -> a
sum []      ~> 0 :: Integer
:t and      ~> and :: [Bool] -> Bool
and []      ~> True :: Bool

```

### Construcción de listas con el operador :

- `x:l` es la lista obtenida añadiendo el elemento `x` al principio de la lista `l`.
- El tipo de `(:)` es `a -> [a] -> [a]`
- El operador `(:)` asocia por la derecha.
- Ejemplos:

```

1:2:3:[] ~> [1,2,3]
1:[2,3]  ~> [1,2,3]

```

### Construcción de listas mediante intervalos numéricos

- Ejemplos de construcción de listas mediante intervalos numéricos:

```

[3..7]    ~> [3,4,5,6,7]
[1,3..10] ~> [1,3,5,7,9]
[7..3]    ~> []
[7,6..3]  ~> [7,6,5,4,3]
[2.5..6.5] ~> [2.5,3.5,4.5,5.5,6.5]
[2.5..6.3] ~> [2.5,3.5,4.5,5.5,6.5]
[2.5..6.7] ~> [2.5,3.5,4.5,5.5,6.5]

```

- `[n..m]` es la lista de los números desde `n` a `m` de 1 en 1.
- `[n,p..m]` es la lista de los números desde `n` a `m` de `d` en `d` donde `d = p-n`.

### Construcción de listas por comprensión

- Ejemplos de lista intensional con un generador:

```

[x*x | x <- [1,3,7] ] ~> [1,9,49]
[2*x | x <- [1..10] ] ~> [2,4,6,8,10,12,14,16,18,20]

```

- Ejemplos de lista intensional con dos generadores:

```

Main> [[x,y] | x <- [1,2], y <- [3,4]]
[[1,3],[1,4],[2,3],[2,4]]

```

- Ejemplos de lista intensional con patrones:

$$|[x+y \mid [x,y] \leftarrow [[1,2],[3,4],[5,6]]] \rightsquigarrow [3,7,11]$$

- Ejemplos de lista intensional con restricciones:

$$|[x \mid (x:y) \leftarrow [[7,2,3],[1,3,4],[9,6]], x > \text{sum } y] \rightsquigarrow [7,9]$$

### Ejemplos de definiciones con listas por comprensión

- `todosPares xs` se verifica si todos los elementos de la lista `xs` son pares. Por ejemplo,

$$\begin{array}{l} \text{todosPares } [2,4,6] \rightsquigarrow \text{True} \\ \text{todosPares } [2,4,6,7] \rightsquigarrow \text{False} \end{array}$$

```
todosPares :: [Int] -> Bool
todosPares xs = (xs == [x | x<-xs, even x])
```

- `triangulares n` es la lista de las listas de números consecutivos desde `[1]` hasta `[1,2,...,n]`. Por ejemplo,

$$|\text{triangulares } 4 \rightsquigarrow [[1],[1,2],[1,2,3],[1,2,3,4]]$$

```
triangulares :: Int -> [[Int]]
triangulares n = [[1..x] | x <- [1..n]]
```

## 1.2. Funciones sobre listas

### Funciones sobre listas

- Patrón para definiciones sobre listas:
  - Base: `[]`
  - Paso: `x:xs`
- Funciones sobre listas del preludio definidas anteriormente:
  - `head l` es la cabeza de la lista `l`
  - `tail l` es el resto de la lista `l`
  - `sum l` es la suma de los elementos de `l`
  - `length l` es el número de elementos de `l`
  - `map f l` es la lista obtenida aplicando `f` a cada elemento de `l`
  - `filter p l` es la lista de los elementos de `l` que cumplen la propiedad `p`

### Comparación y ordenación lexicográfica de listas

- Se pueden comparar y ordenar listas (con operadores como == y <) con la condición de que se puedan o comparar y ordenar sus elementos.
- Definición de la igualdad de listas:

```

Prelude
[]      == []      = True
(x:xs) == (y:ys) = x==y && xs==ys
_      == _      = False

```

### Concatenación de listas

- `l1 ++ l2` es la concatenación de `l1` y `l2`. Por ejemplo,

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

```

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

```

- `concat l` es la concatenación de las lista de `l`. Por ejemplo,

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

```

Prelude
concat :: [[a]] -> [a]
concat []      = []
concat (x:xs) = x ++ concat xs

```

### Propiedades de la concatenación de listas

- Longitud de la concatenación:

```

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.

```

- Longitud de la concatenación general:



```
prop_length_concat :: [[Int]] -> Bool
prop_length_concat xss =
  length(concat xss) == sum (map length xss)
```

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

## Selección de partes de una lista

- `head 1` es la cabeza de la lista 1. Por ejemplo,

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

Prelude

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

- `tail 1` es el resto de la lista 1. Por ejemplo,

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

Prelude

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

- Conjetura:

```
prop_head_tail_1 :: [Int] -> Bool
prop_head_tail_1 xs =
  (head xs) : (tail xs) == xs
```

- Comprobación de la conjetura:

```
| quickCheck prop_head_tail_1
Falsifiable, after 2 tests:
[]
```

- Comprobación del contraejemplo:

```
Main> (head []) : (tail []) == []
False
Main> head []
Program error: pattern match failure: head []
```

- Propiedad:

```
prop_head_tail :: [Int] -> Property
prop_head_tail xs =
  not (null xs) ==> (head xs) : (tail xs) == xs
```

- Comprobación de la propiedad:

```
|Main> quickCheck prop_head_tail
OK, passed 100 tests.
```

- `last l` es el último elemento de la lista `l`. Por ejemplo,

```
|last [1,2,3] ~> 3
```

Prelude

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

- `init l` es la lista `l` sin el último elemento. Por ejemplo,

```
|init [1,2,3] ~> [1,2]
|init [4] ~> []
```

Prelude

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

- Propiedad:

```
prop_init_last :: [Int] -> Property
prop_init_last xs =
  not (null xs) ==>
    (init xs) ++ [last xs] == xs
```

- Comprobación:

```
|Main> quickCheck prop_init_last
OK, passed 100 tests.
```

- `take n l` es la lista de los `n` primeros elementos de `l`. Por ejemplo,

```
take 2 [3,5,4,7] ~> [3,5]
take 12 [3,5,4,7] ~> [3,5,4,7]
```

Prelude

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

- Propiedad:

```
prop_take :: Int -> Int -> [Int] -> Property
prop_take n m xs =
  n >= 0 && m >= 0 ==>
  take n (take m xs) == take (min n m) xs
```

- Comprobación:

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

- `drop n l` es la lista obtenida eliminando los primeros `n` elementos de la lista `l`. Por ejemplo,

```
drop 2 [3..10] ~> [5,6,7,8,9,10]
drop 12 [3..10] ~> []
```

Prelude

```
drop :: Int -> [a] -> [a]
drop n xs | n <= 0 = xs
drop _ []         = []
drop n (_:xs)     = drop (n-1) xs
```

- Propiedad de `drop`:

```
prop_drop :: Int -> Int -> [Int] -> Property
prop_drop n m xs =
  n >= 0 && m >= 0 ==>
  drop n (drop m xs) == drop (n+m) xs
```

- Propiedad de `take` y `drop`:

```
prop_take_drop :: Int -> [Int] -> Bool
prop_take_drop n xs =
  (take n xs) ++ (drop n xs) == xs
```

- `l !! n` es elemento  $n$ -ésimo de `l`, empezando a numerar con el 0. Por ejemplo,

```
| [1,3,2,4,9,7] !! 3 ~ 4
```

```
----- Prelude -----
infixl 9 !!

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

### Inversa de una lista

- `reverse l` es la inversa de `l`. Por ejemplo,

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

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

- Propiedades:

```
prop_reverse :: [Int] -> [Int] -> Bool
prop_reverse xs ys =
  reverse(xs ++ ys) == (reverse ys) ++ (reverse xs)
  && reverse(reverse xs) == xs
```

### Longitud de una lista

- `length l` es el número de elementos de `l`. Por ejemplo,

```
| length [1,3,6] ~ 3
```

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

## Test de pertenencia a una lista

- `elem e l` se verifica si `e` es un elemento de `l`. Por ejemplo,

```
elem 2 [1,2,3] ~> True
elem 4 [1,2,3] ~> False
```

- Definición recursiva

```
elem_1 :: Eq a => a -> [a] -> Bool
elem_1 _ []      = False
elem_1 x (y:ys) = (x==y) || elem_1 x ys
```

- Definición con `or` y lista por comprensión:

```
elem_2 :: Eq a => a -> [a] -> Bool
elem_2 x ys = or [x==y | y <- ys]
```

- `notElem e l` se verifica si `e` es un elemento de `l`. Por ejemplo,

```
notElem 2 [1,2,3] ~> False
notElem 4 [1,2,3] ~> True
```

- Definición recursiva

```
notElem_1 :: Eq a => a -> [a] -> Bool
notElem_1 _ []      = True
notElem_1 x (y:ys) = (x/=y) && notElem_1 x ys
```

- Definición con `and` y lista por comprensión:

```
notElem_2 :: Eq a => a -> [a] -> Bool
notElem_2 x ys = and [x/=y | y <- ys]
```

- Equivalencia de las definiciones:

```
prop_equivalencia_notElem :: Int -> [Int] -> Bool
prop_equivalencia_notElem x ys =
  notElem_1 x ys == notElem x ys &&
  notElem_2 x ys == notElem x ys

prop_equivalencia_elem :: Int -> [Int] -> Bool
prop_equivalencia_elem x ys =
  elem_1 x ys == elem x ys &&
  elem_2 x ys == elem x ys
```

- Propiedad de `elem` y `notElem`:

```
prop_elem_notElem :: Int -> [Int] -> Bool
prop_elem_notElem x ys =
  (elem x ys || notElem x ys) &&
  not ((elem x ys) && (notElem x ys))
```

### 1.3. Funciones de orden superior sobre listas

#### Funciones de orden superior sobre listas: `map`

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

```
|map (2*) [1,2,3] ~> [2,4,6]
```

- Definición por comprensión:

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

- Definición recursiva:

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

- `filter p l` es la lista de los elementos de `l` 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 por comprensión:

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

- Definición por recursión:

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

## Ejemplo de abstracción de patrones

- Ejemplos de definiciones con el mismo patrón:

- `sum l` es la suma de los elementos de `l`. Por ejemplo,

```
sum [3,4,5,6] ~> 18
```

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

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

- `product l` es el producto de los elementos de `l`. Por ejemplo,

```
product [2,3,5] ~> 30
```

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

```
product [] = 1
product (x:xs) = x * product xs
```

- Ejemplos de definiciones con el mismo patrón:

- `and l` se verifica si todos los elementos de `l` son verdaderos. Por ejemplo,

```
and [1<2, 2<3, 1 /= 0] ~> True
```

```
and [1<2, 2<3, 1 == 0] ~> False
```

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

```
and [] = True
and (x:xs) = x && and xs
```

- `or l` se verifica si algún elemento de `l` es verdadero. Por ejemplo,

```
or [2>3, 5<9] ~> True
```

```
or [2>3, 9<5] ~> False
```

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

```
or [] = False
or (x:xs) = x || or xs
```

- Patrón:

`foldr op e l` pliega por la derecha la lista `l` colocando el operador `op` entre sus elementos y el elemento `e` al final. Es decir,

```
foldr f e [x1,x2,x3] ~> x1 f (x2 f (x3 f e))
```

```
foldr f e [x1,x2,...,xn] ~> x1 f (x2 f (... f (xn f e)))
```

Por ejemplo,

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

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

```

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

```

- Redefiniciones con el patrón:

```

sum_1      = foldr (+) 0
product_1  = foldr (*) 1
and_1      = foldr (&&) True
or_1       = foldr (||) False

```

- Definición del factorial mediante plegados:

`fact n` es el factorial de `n`. Por ejemplo,

```
| fact 5 ~> 120
```

```
fact n = foldr (*) 1 [1..n]
```

- Plegado por la izquierda:

`foldl op e l` pliega por la izquierda la lista `l` colocando el operador `op` entre sus elementos y el elemento `e` al principio. Es decir,

```

foldl f e [x1,x2,x3]    ~> (((e f x1) f x2) f x3
foldl f e [x1,x2,...,xn] ~> (...((e f x1) f x2) ... f xn

```

Por ejemplo,

```

| foldl (+) 6 [2,3,5] ~> ((6+2)+3)+5 ~> 16
| foldl (-) 3 [2,3,5] ~> ((6-2)-3)-5 ~> -4

```

```

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

```

### Plegado acumulativo: `scanr`

- `scanr op e l` pliega por la derecha la lista `l` colocando el operador `op` entre sus elementos y el elemento `e` al final y escribe los resultados acumulados. Por ejemplo,



```
scanr (+) 6 [2,3,5] ~> [2+(3+(5+6)),3+(5+6),5+6,6]
                    ~> [16,14,11,6]
scanr (-) 6 [2,3,5] ~> [2-(3-(5-6)),3-(5-6),5-6,6]
                    ~> [-2,4,-1,6]
```

Prelude

```
scanr :: (a -> b -> b) -> b -> [a] -> [b]
scanr f e []      = [e]
scanr f e (x:xs) = (f x q) : (q:qs)
                  where (q:qs) = scanr f e xs
```

- `scanl` es análogo a `scanr` pero empezando por la izquierda. Por ejemplo,

```
scanl (+) 3 [2,3,5] ~> [3,5,8,13]
```

- `factoriales n` es la lista de los factoriales desde el factorial de 0 hasta el factorial de `n`. Por ejemplo,

```
factoriales 5 ~> [1,1,2,6,24,120]
```

```
factoriales :: Int -> [Int]
factoriales n = scanl (*) 1 [1..n]
```

### Segmentos iniciales: `takeWhile`

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

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

Prelude

```
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p []      = []
takeWhile p (x:xs)
  | p x      = x : takeWhile p xs
  | otherwise = []
```

### Segmentos finales: `dropWhile`

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

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

```

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

```

## 1.4. Ordenación de listas

### Ordenación por inserción

- `inserta e l` inserta el elemento `e` en la lista `l` delante del primer elemento de `l` 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

```

- `ordena_por_inserción l` es la lista `l` ordenada mediante inserción, Por ejemplo,

```
| ordena_por_inserción [2,4,3,6,3] ~> [2,3,3,4,6]
```

- Definición recursiva:

```

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

```

- Definición por plegado por la derecha:

```

ordena_por_inserción_1 :: Ord a => [a] -> [a]
ordena_por_inserción_1 = foldr inserta []

```

- Definición por plegado por la izquierda:

```

ordena_por_inserción_2 :: Ord a => [a] -> [a]
ordena_por_inserción_2 = foldl (flip inserta) []

```

- La última es la más eficiente:

```
ordena_por_inserción [100,99..1] (51959 reductions, 68132 cells)
ordena_por_inserción_1 [100,99..1] (51960 reductions, 68034 cells)
ordena_por_inserción_2 [100,99..1] ( 3451 reductions,  5172 cells)
```

- mínimo l es el menor elemento de la lista l. Por ejemplo,

```
mínimo [3,2,5] ~> 2
```

```
mínimo l = head (ordena_por_inserción l)
```

La complejidad de mínimo es lineal:

```
mínimo [10,9..1]           (   300 red)
mínimo [100,99..1]        (  2550 red)
mínimo [1000,999..1]     ( 25050 red)
mínimo [10000,9999..1]  (250050 red)
```

aunque la complejidad de ordena\_por\_inserción es cuadrática

```
ordena_por_inserción [10,9..1]           (   750 red)
ordena_por_inserción [100,99..1]        (  51960 red)
ordena_por_inserción [1000,999..1]     (5019060 red)
ordena_por_inserción [10000,9999..1]  (500190060 red)
```

- lista\_ordenada l se verifica si la lista l está ordenada de menor a mayor. Por ejemplo,

```
lista_ordenada [1,3,3,5] ~> True
lista_ordenada [1,3,5,3] ~> False
```

```
lista_ordenada :: Ord a => [a] -> Bool
lista_ordenada []      = True
lista_ordenada [_]    = True
lista_ordenada (x:y:xs) = x<=y && lista_ordenada (y:xs)
```

- El valor de ordena\_por\_inserción es una lista ordenada

```
prop_ordena_por_inserción_ordenada :: [Int] -> Bool
prop_ordena_por_inserción_ordenada xs =
  lista_ordenada (ordena_por_inserción xs)
```

### Ordenación por mezcla

- `mezcla l1 l2` es la lista ordenada obtenida al mezclar las listas ordenadas `l1` y `l2`. Por ejemplo,

```
|mezcla [1,3,5] [2,9] ~> [1,2,3,5,9]
```

```
mezcla :: Ord a => [a] -> [a] -> [a]
mezcla [] ys      = ys
mezcla xs []      = xs
mezcla (x:xs) (y:ys)
  | x <= y        = x : mezcla xs (y:ys)
  | otherwise     = y : mezcla (x:xs) ys
```

- `ordena_por_m l` es la lista `l` ordenada mediante mezclas, Por ejemplo,

```
|ordena_por_m [2,4,3,6,3] ~> [2,3,3,4,6]
```

```
ordena_por_m :: Ord a => [a] -> [a]
ordena_por_m [] = []
ordena_por_m [x] = [x]
ordena_por_m xs =
  mezcla (ordena_por_m ys) (ordena_por_m zs)
  where medio = (length xs) `div` 2
        ys    = take medio xs
        zs    = drop medio xs
```

- El valor de `ordena_por_m` es una lista ordenada

```
prop_ordena_por_mezcla_ordenada :: [Int] -> Bool
prop_ordena_por_mezcla_ordenada xs =
  lista_ordenada (ordena_por_m xs)
```

### Ordenación rápida (“quicksort”)

`ordenaR xs` es la lista `xs` ordenada mediante el procedimiento de ordenación rápida. Por ejemplo,

```
|ordenaR [5,2,7,7,5,19,3,8,6] ~> [2,3,5,5,6,7,7,8,19]
```

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

## 2. Listas especiales

### 2.1. Cadenas

#### Cadenas

- Las **cadenas** son listas cuyos elementos son caracteres.
- Las cadenas se anotan entre comillas.
- Ejemplos de cadenas: "lunes", "Juan Luis"
- A las cadenas se le pueden aplicar las funciones sobre listas. Por ejemplo,

```
length "martes"  ~> 6
"mar" ++ "tes"   ~> "martes"
"hola" < "mundo" ~> True
```

- El tipo de las cadenas es **String**.

### 2.2. Caracteres

#### Caracteres

- Los **caracteres** pueden ser letras, cifras y signos de puntuación.
- Los caracteres se anotan entre apóstrofes.
- Ejemplos de caracteres: 'a', '+'
- El tipo de los caracteres es **Char**.
- Una lista de caracteres es una cadena. Por ejemplo,

```
['h', 'o', 'l', 'a'] ~> "hola"
head "hola"          ~> 'h'
```

### 2.3. Funciones de cadenas y caracteres

#### Funciones de transformación entre cadenas y enteros

- **ord c** es el código ASCII del carácter *c*.
- **chr n** es el carácter de código ASCII el entero *n*.
- Ejemplos:

```
| :load Hugs.Char
| ord 'A' ~> 65
| chr 65 ~> 'A'
```

- Para usar las funciones sobre caracteres en programas hay que importar el módulo Char escribiendo al principio del fichero

```
import Hugs.Char
```

### Funciones reconocedoras predefinidas

- Funciones:
  - `isSpace x`      x es un espacio
  - `isUpper x`      x está en mayúscula
  - `isLower x`      x está en minúscula
  - `isAlpha x`      x es un carácter alfabético
  - `isDigit x`      x es un dígito
  - `isAlphaNum x`   x es un carácter alfanumérico
- Ejemplos de definición:

```
----- Prelude -----
isDigit c = c >= '0' && c <= '9'
```

### Funciones sobre caracteres

- `dígitoDeCarácter c` es el dígito correspondiente al carácter numérico c. Por ejemplo,

```
| dígitoDeCarácter '3' ~> 3
```

```
dígitoDeCarácter :: Char -> Int
dígitoDeCarácter c = ord c - ord '0'
```

#### Definición alternativa

```
dígitoDeCarácter' :: Char -> Int
dígitoDeCarácter' c
  | isDigit c = ord c - ord '0'
```

- `carácterDeDígito n` es el carácter correspondiente al dígito n. Por ejemplo,

```
| carácterDeDígito 3 ~> '3'
```

```
carácterDeDígito :: Int -> Char
carácterDeDígito n = chr (n + ord '0')
```

**Convesiones entre minúsculas y mayúsculas:**

- `toUpper` `c` es el carácter `c` en mayúscula.
- `toLower` `c` es el carácter `c` en minúscula.
- Ejemplos:

```
toUpper 'a'           ~> 'A'
toLower 'A'          ~> 'a'
map toUpper "sevilla" ~> "SEVILLA"
map toLower "SEVILLA" ~> "sevilla"
```

**Funciones del preludio específicas para cadenas**

- `words` `c` es la lista de palabras de la cadena `c`. Por ejemplo,

```
Main> words "esto es una cadena"
["esto", "es", "una", "cadena"]
```

- `lines` `c` es la lista de líneas de la cadena `c`. Por ejemplo,

```
Main> lines "primera linea\ny segunda"
["primera linea", "y segunda"]
Main> words "primera linea\ny segunda"
["primera", "linea", "y", "segunda"]
```

- `unwords` es la inversa de `words`. Por ejemplo,

```
Main> unwords ["esto", "es", "una", "cadena"]
"esto es una cadena"
```

- `unlines` es la inversa de `lines`. Por ejemplo,

```
Main> unlines ["primera linea", "y segunda"]
"primera linea\ny segunda\n"
```

**2.4. Listas infinitas y evaluación perezosa****Listas infinitas**

- `desde n` es la lista de los números enteros a partir de `n`. Por ejemplo,

```
| desde 5 ~> [5,6,7,8,9,10,11,12,13,14, {Interrupted!}]
```

se interrumpe con Control-C.

```
desde :: Int -> [Int]
desde n = n : desde (n+1)
```

Definición alternativa

```
desde' :: Int -> [Int]
desde' n = [n..]
```

### Cálculos con listas infinitas

- Calcular la lista de los 10 primeros sucesores de 7:

```
Main> tail (take 11 (desde 7))
[8,9,10,11,12,13,14,15,16,17]
Main> tail (take 11 [7..])
[8,9,10,11,12,13,14,15,16,17]
```

- Calcular las potencias de 2 menores que 1000:

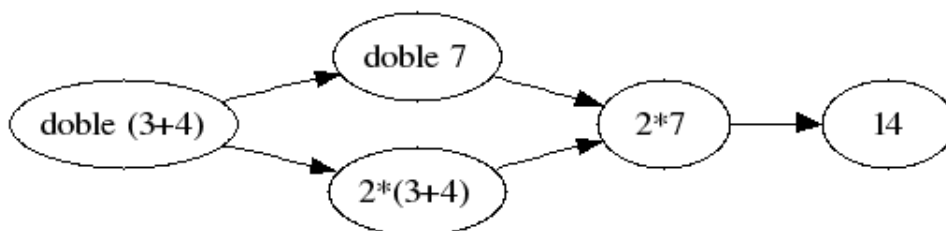
```
Main> takeWhile (<1000) (map (2^) (desde 1))
[2,4,8,16,32,64,128,256,512]
Main> takeWhile (<1000) (map (2^) [1..])
[2,4,8,16,32,64,128,256,512]
```

### Evaluación perezosa

Se considera la siguiente definición

```
doble :: Int -> Int
doble x = 2*x
```

Entonces, las posibles evaluaciones de `doble (3+4)` son



Se considera la siguiente definición

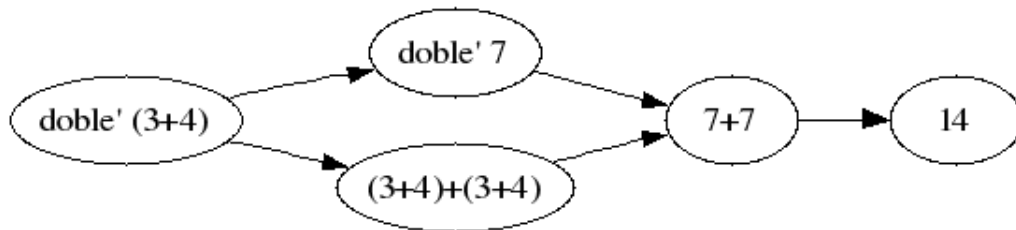


```

doble' :: Int -> Int
doble' x = x+x

```

Entonces, las posibles evaluaciones de `doble' (3+4)` son



Notar que en la rama superior, `3+4` sólo se evalúa una vez.

### Métodos de evaluación

- **evaluación perezosa** (en inglés “lazy evaluation”): se calcula una expresión parcial si realmente se necesita el valor para calcular el resultado.
- **evaluación voraz** (en inglés “eager evaluation”): se calcula el valor de los parámetros y se aplica la función a sus valores.
- Haskell usa evaluación perezosa.
- LISP, Scheme y ML usan evaluación voraz.

### Ejemplo de ventaja de la evaluación perezosa

En el tema anterior, se definió `primo` mediante

```
primo x = divisores x == [1,x]
```

Al aplicar la definición sólo se calculan los dos primeros primos. Por ejemplo,

```

ED> :set +s
ED> primo 30
False
(108 reductions, 177 cells)
ED> primo 30000
False
(108 reductions, 177 cells)

```

### Explicación del comportamiento perezoso de `==` y `&&`

- La regla recursiva de la definición de `==` es

$$(x:xs) == (y:ys) = x==y \ \&\& \ xs==ys$$

Si `x==y` es falsa, entonces también lo es `(x:xs) == (y:ys)`.

- La definición de `&&` es

Prelude

```
False && x = False
True  && x = x
```

Si el primer parámetro es falso, entonces la conjunción es falsa.

### Cálculos infinitos

Las funciones que necesitan todos los elementos de una lista, no pueden aplicarse a listas infinitas. Por ejemplo,

```
Main> length (desde 1)
ERROR - Garbage collection fails to reclaim sufficient space

Main> head (desde 1)
1

Main> (desde 1) ++ [3]
[1,2,3,4,5,6,7,8,9,10,11,12,...Interrupted!

Main> last (desde 1)
ERROR - Garbage collection fails to reclaim sufficient space
```

### Funciones sobre listas infinitas definidas en el prelude

- `[n..]` es equivalente a `desde n`.
- `repeat x` es una lista infinita con el único elemento `x`. Por ejemplo,

```
Main> repeat 'a'
"aaaaaaaaaaaaaaaaaaaaaaaaaa{Interrupted!}
```

```
repeat' :: a -> [a]
repeat' x = x : repeat' x
```

Una definición alternativa es

---

 Prelude
 

---

```
repeat :: a -> [a]
repeat x = xs where xs = x:xs
```

- `replicate n x` es una lista con  $n$  copias del elemento  $x$ . Por ejemplo,

```
Main> replicate 10 3
[3,3,3,3,3,3,3,3,3,3]
```

---

 Prelude
 

---

```
replicate :: Int -> a -> [a]
replicate n x = take n (repeat x)
```

- `iterate f x` es la lista cuyo primer elemento es  $x$  y los siguientes elementos se calculan aplicando la función  $f$  al elemento anterior. Por ejemplo,

```
Main> iterate (+1) 3
[3,4,5,6,7,8,9,10,11,12,{Interrupted!}]
Main> iterate (*2) 1
[1,2,4,8,16,32,64,{Interrupted!}]
Main> iterate ('div' 10) 1972
[1972,197,19,1,0,0,0,0,0,0,{Interrupted!}]
```

---

 Prelude
 

---

```
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)
```

### Presentación de un número como cadena

- `deEnteroACadena n` es la cadena correspondiente al número entero  $n$ . Por ejemplo, `deEnteroACadena 1972`  $\rightsquigarrow$  "1972"

```
deEnteroACadena :: Int -> String
deEnteroACadena = map carácterDeDígito
                  . reverse
                  . map ('rem' 10)
                  . takeWhile (/= 0)
                  . iterate ('div' 10)
```

### Ejemplo de cálculo

```
iterate ('div' 10) 1972            $\rightsquigarrow$  [1972,197,19,1,0,0,0,...]
(takeWhile (/= 0) . iterate ('div' 10)) 1972  $\rightsquigarrow$  [1972,197,19,1]
map ('rem' 10) [1972,197,19,1]    $\rightsquigarrow$  [2,7,9,1]
reverse [2,7,9,1]                $\rightsquigarrow$  [1,9,7,2]
map carácterDeDígito [1,9,7,2]   $\rightsquigarrow$  "1972"
```

- La función `deEnteroACadena` es un caso particular de `show`. Por ejemplo,

```
| show 1972 ~> "1972"
```

### Cálculo perezoso de la lista de los números primos

- `primos_por_criba` es la lista de los números primos mediante la criba de Eratóstenes.

```
Main> primos_por_criba
[2,3,5,7,11,13,17,19,23,29,{Interrupted!}]
Main> take 10 primos_por_criba
[2,3,5,7,11,13,17,19,23,29]
```

```
primos_por_criba :: [Int]
primos_por_criba = map head (iterate eliminar [2..])
  where eliminar (x:xs) = filter (no_multiplo x) xs
        no_multiplo x y = y `mod` x /= 0
```

- Para ver el cálculo, consideramos la siguiente variación

```
primos_por_criba_aux =
  map (take 10) (iterate eliminar [2..])
  where eliminar (x:xs) = filter (no_multiplo x) xs
        no_multiplo x y = y `mod` x /= 0
```

Entonces,

```
Main> take 5 primos_por_criba_aux
[[ 2, 3, 4, 5, 6, 7, 8, 9,10,11],
 [ 3, 5, 7, 9,11,13,15,17,19,21],
 [ 5, 7,11,13,17,19,23,25,29,31],
 [ 7,11,13,17,19,23,29,31,37,41],
 [11,13,17,19,23,29,31,37,41,43]]
```

- Definición con lista por comprensión:

```
primos_por_criba' :: [Int]
primos_por_criba' = criba [2..]
  where criba (p:xs) = p : criba [n | n<-xs,
                                     n `mod` p /= 0]
```

## 3. Tuplas

### 3.1. Uso de tuplas

#### Tuplas

- Una **tupla** consiste en un número fijo de valores, que están agrupados como una entidad.
- Ejemplos de uso de tuplas en modelizaciones:
 

punto	(3.5,4.2)	:: (Float, Float)
teléfono	("Ana Pi", 954213465)	:: (String, Int)
precio	("periódico", 1.9)	:: (String, Float)
vacía	()	:: ()
- Los valores pueden ser de diferentes tipos.
- Las tuplas se anotan con paréntesis.
- Clases de tuplas:
  - Un **par** es una tupla con dos elementos.
  - Una **terna** es una tupla con tres elementos.
  - La **tupla vacía** es la tupla sin elementos.

#### Funciones del prelude sobre tuplas

- **fst p** es la primera componente del par p. Por ejemplo,

|fst (3,2) ~> 3

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

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

- **snd p** es la segunda componente del par p. Por ejemplo,

|snd (3,2) ~> 2

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

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

**Definición de funciones sobre tuplas**

- `fst3 t` es la primera componente de la terna `t`.
- `snd3 t` es la segunda componente de la terna `t`.
- `thd3 t` es la tercera componente de la terna `t`.
- Ejemplos:

```
fst3 (3,2,5) ~> 3
snd3 (3,2,5) ~> 2
thd3 (3,2,5) ~> 5
```

```
fst3 :: (a,b,c) -> a
fst3 (x,_,_) = x
snd3 :: (a,b,c) -> b
snd3 (_,y,_) = y
thd3 :: (a,b,c) -> c
thd3 (_,_,z) = z
```

- `variable p` es la cadena correspondiente al par `p` formado por un carácter y un número. Por ejemplo,

```
variable ('x',3) ~> "x3"
```

```
variable (c,n) = [c] ++ show n
```

```
Main> :type variable
variable :: Show a => (Char,a) -> [Char]
```

- `distanciaL p` es la distancia del punto `p`, representado mediante listas, al origen. Por ejemplo,

```
distanciaL [3.0,4.0] ~> 5.0
```

```
distanciaL :: [Float] -> Float
distanciaL [x,y] = sqrt (x*x+y*y)
```

- `distanciaT p` es la distancia del punto `p`, representado mediante tuplas, al origen. Por ejemplo,

```
distanciaT (3.0,4.0) ~> 5.0
```

```
distanciaT :: (Float, Float) -> Float
distanciaT (x,y) = sqrt (x*x+y*y)
```

### Ejemplo de uso de tupla para devolver varios resultados

- `splitAt n l` es el par formado por la lista de los `n` primeros elementos de la lista `l` y la lista `l` sin los `n` primeros elementos. Por ejemplo,

```
splitAt 3 [5,6,7,8,9,2,3] ~> ([5,6,7],[8,9,2,3])
splitAt 4 "sacacorcho"    ~> ("saca","corcho")
```

Prelude

```
splitAt :: Int -> [a] -> ([a], [a])
splitAt n xs | n <= 0 = ([],xs)
splitAt _ []         = ([],[])
splitAt n (x:xs)     = (x:xs',xs'')
  where (xs',xs'') = splitAt (n-1) xs
```

- Definición alternativa

```
splitAt_alt :: Int -> [a] -> ([a], [a])
splitAt_alt n xs = (take n xs, drop n xs)
```

- La definición alternativa es un poco menos eficiente, Por ejemplo,

```
Main> :s +s
Main> splitAt_alt 3 [5,6,7,8,9,2,3]
([5,6,7],[8,9,2,3])
(228 reductions, 330 cells)
Main> splitAt 3 [5,6,7,8,9,2,3]
([5,6,7],[8,9,2,3])
(176 reductions, 283 cells)
```

### Ejemplo de uso de tupla para aumentar la eficiencia

- `incmin l` es la lista obtenida añadiendo a cada elemento de `l` el menor elemento de `l`. Por ejemplo,

```
incmin [3,1,4,1,5,9,2,6] ~> [4,2,5,2,6,10,3,7]
```

```
incmin :: [Int] -> [Int]
incmin l = map (+e) l
  where e = mínimo l
        mínimo [x] = x
        mínimo (x:y:xs) = min x (mínimo (y:xs))
```

- Con la definición anterior se recorre la lista dos veces: una para calcular el mínimo y otra para sumarlo.
- Con la siguiente definición la lista se recorre sólo una vez.

```

incmin' :: [Int] -> [Int]
incmin' [] = []
incmin' l = nuevalista
  where (minv, nuevalista) = un_paso l
        un_paso [x]      = (x, [x+minv])
        un_paso (x:xs)  = (min x y, (x+minv):ys)
                          where (y,ys) = un_paso xs

```

## 3.2. Listas y tuplas

### Listas de asociación

- Una **lista de asociación** es una lista de pares. Los primeros elementos son las **claves** y los segundos son los **valores**.
- `buscar z l` es el valor del primer elemento de la lista de asociación `l` cuya clave es `z`. Por ejemplo,

```

| buscar 'b' [('a',1),('b',2),('c',3),('b',4)] ~> 2

```

```

buscar :: Eq a => a -> [(a,b)] -> b
buscar z ((x,y):r)
  | x == z    = y
  | otherwise = buscar z r

```

### Funciones de agrupamiento

- `zip x y` es el producto cartesiano de `x` e `y`. Por ejemplo,

```

| zip [1,2,3] "abc" ~> [(1,'a'),(2,'b'),(3,'c')]
| zip [1,2] "abc"   ~> [(1,'a'),(2,'b')]

```

- `zipWith f x y` es la lista obtenida aplicando la función `f` a los elementos correspondientes de las listas `x` e `y`. Por ejemplo,

```

| zipWith (+) [1,2,3] [4,5,6] ~> [5,7,9]
| zipWith (*) [1,2,3] [4,5,6] ~> [4,10,18]

```



```

                                Prelude
zipWith :: (a->b->c) -> [a]->[b]->[c]
zipWith f (a:as) (b:bs) = f a b : zipWith f as bs
zipWith _ _ _ = []

zip :: [a] -> [b] -> [(a,b)]
zip = zipWith (\a b -> (a,b))

```

- Definición alternativa de zip sin función anónima:

```

zip_alt_1 = zipWith emparejar
  where emparejar a b = (a,b)

```

- Definición recursiva de zip:

```

zip_alt_2 (a:as) (b:bs) = (a,b) : zip_alt_2 as bs
zip_alt_2 _ _ = []

```

### Definiciones con zip: Posiciones

- `posiciones xs` es la lista de los pares formados por los elementos de `xs` junto con su posición. Por ejemplo,

```

|posiciones [3,5,2,5] ~> [(3,1), (5,2), (2,3), (5,4)]

```

```

posiciones :: [a] -> [(a,Int)]
posiciones xs = zip xs [1..length xs]

```

- `posición x ys` es la posición del elemento `x` en la lista `ys`. Por ejemplo,

```

|posición 5 [1,5,3] ~> 2

```

```

posición :: Eq a => a -> [a] -> Int
posición x xs = buscar x (posiciones xs)

```

### Definiciones con zip: Fibonacci

- `fibs` es la sucesión de los números de Fibonacci. Por ejemplo,

```

|take 10 fibs ~> [1,1,2,3,5,8,13,21,34,55]

```

```

fibs = 1 : 1 : [a+b | (a,b) <- zip fibs (tail fibs)]

```

### 3.3. Tuplas y currificación

#### Formas cartesiana y currificada

- Una función está en **forma cartesiana** si su argumento es una tupla. Por ejemplo,

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

- En cambio, la función

```
suma_currificada :: Int -> Int -> Int
suma_currificada x y = x+y
```

está en **forma currificada**.

- curry f** es la versión currificada de la función f. Por ejemplo,

```
| curry suma_cartesiana 2 3 ~> 5
```

```
----- Prelude -----
curry :: ((a,b) -> c) -> (a -> b -> c)
curry f x y = f (x,y)
```

- uncurry f** es la versión cartesiana de la función f. Por ejemplo,

```
| uncurry suma_currificada (2,3) ~> 5
```

```
----- Prelude -----
uncurry :: (a -> b -> c) -> ((a,b) -> c)
uncurry f p = f (fst p) (snd p)
```

## 4. Tipos de datos definidos

### 4.1. Definiciones de tipos

#### Definiciones de tipos con **type**

- Un punto es un par de números reales. Por ejemplo,

```
| (3.0,4.0) :: Punto
```

```
type Punto = (Float, Float)
```

- `distancia_al_origen p` es la distancia del punto `p` al origen. Por ejemplo,

```
| distancia_al_origen (3,4) ~> 5.0
```

```
distancia_al_origen :: Punto -> Float
distancia_al_origen (x,y) = sqrt (x*x+y*y)
```

- `distancia p1 p2` es la distancia entre los puntos `p1` y `p2`. Por ejemplo,

```
| distancia (2,4) (5,8) ~> 5.0
```

```
distancia :: Punto -> Punto -> Float
distancia (x,y) (x',y') = sqrt((x-x')^2+(y-y')^2)
```

- Un camino es una lista de puntos. Por ejemplo,

```
| [(1,2),(4,6),(7,10)] :: Camino
```

```
type Camino = [Punto]
```

- `longitud_camino c` es la longitud del camino `c`. Por ejemplo,

```
| longitud_camino [(1,2),(4,6),(7,10)] ~> 10.0
```

- Definición recursiva:

```
longitud_camino :: Camino -> Float
longitud_camino [] = 0
longitud_camino (p:q:xs) =
    (distancia p q) + (longitud_camino (q:xs))
```

- Definición no recursiva:

```
longitud_camino' :: Camino -> Float
longitud_camino' xs =
    sum [distancia p q |
        (p,q) <- zip (init xs) (tail xs)]
```

Evaluación paso a paso:

```
longitud_camino [(1,2),(4,6),(7,10)]
= sum [distancia p q | (p,q) <- zip (init [(1,2),(4,6),(7,10)])
                                (tail [(1,2),(4,6),(7,10)])]
= sum [distancia p q | (p,q) <- zip [(1,2),(4,6)] [(4,6),(7,10)]]
= sum [distancia p q | (p,q) <- [(1,2),(4,6)],[(4,6),(7,10)]]
= sum [5.0,5.0]
= 10
```

## 4.2. Números racionales

### Números racionales

- En esta sección se presentan los números racionales como una aplicación de tuplas y definición de tipos.
- Un número racional es un par de enteros

```
type Racional = (Int, Int)
```

- Ejemplos de números racionales:

```
qCero   = (0,1)
qUno    = (1,1)
qDos    = (2,1)
qTres   = (3,1)
qMedio  = (1,2)
qTercio = (1,3)
qCuarto = (1,4)
```

- `simplificar x` es el número racional `x` simplificado. Por ejemplo,

```
simplificar (12,24)  ~> (1,2)
simplificar (12,-24) ~> (-1,2)
simplificar (-12,-24) ~> (1,2)
simplificar (-12,24) ~> (-1,2)
```

```
simplificar (n,d) =
  (((signum d)*n) 'div' m, (abs d) 'div' m)
  where m = gcd n d
```

- `gcd x y` es el máximo común divisor de `x` e `y`. Por ejemplo,

```
gcd 6 15 ~> 3
```

```
----- Prelude -----
gcd 0 0 = error "Prelude.gcd: gcd 0 0 is undefined"
gcd x y = gcd' (abs x) (abs y)
          where gcd' x 0 = x
                gcd' x y = gcd' y (x 'rem' y)
```

- Definición alternativa

```
gcd_alt' x y = head [z | z <- [a,a-1..1], divisible x z,
                      divisible y z]
  where a          = min (abs x) (abs y)
        divisible x y = x `rem` y == 0
```

- Operaciones con números racionales. Por ejemplo,

```
qMul (1,2) (2,3) ~> (1,3)
qDiv (1,2) (1,4) ~> (2,1)
qSum (1,2) (3,4) ~> (5,4)
qRes (1,2) (3,4) ~> (-1,4)
```

```
qMul, qDiv, qSum, qRes :: Racional -> Racional -> Racional
qMul (x1,y1) (x2,y2) = simplificar (x1*x2, y1*y2)
qDiv (x1,y1) (x2,y2) = simplificar (x1*y2, y1*x2)
qSum (x1,y1) (x2,y2) = simplificar (x1*y2+y1*x2, y1*y2)
qRes (x1,y1) (x2,y2) = simplificar (x1*y2-y1*x2, y1*y2)
```

- escribeRacional x es la cadena correspondiente al número racional x. Por ejemplo,

```
escribeRacional (10,12) ~> "5/6"
escribeRacional (12,12) ~> "1"
escribeRacional (qMul (1,2) (2,3)) ~> "1/3"
```

```
escribeRacional :: Racional -> String
escribeRacional (x,y)
  | y' == 1    = show x'
  | otherwise = show x' ++ "/" ++ show y'
  where (x',y') = simplificar (x,y)
```

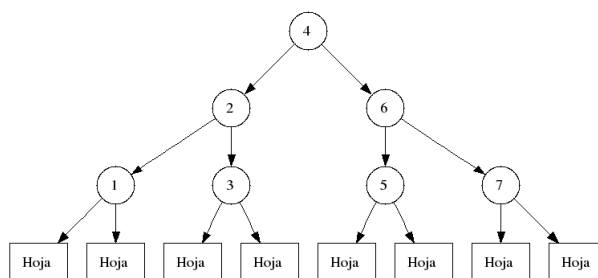
### 4.3. Tipo de dato definido: Árboles

#### Tipo de dato definido: Árboles

- Un **árbol** de tipo a es una hoja de tipo a o es un nodo de tipo a con dos hijos que son árboles de tipo a.

```
data Árbol a = Hoja
             | Nodo a (Árbol a) (Árbol a)
             deriving Show
```

- El nombre del tipo de datos es `Árbol`
  - Las funciones constructoras son `Hoja` y `Nodo`
- El nombre del tipo de dato y de sus funciones constructoras tienen que empezar por mayúscula.
  - Ejemplo de árbol: El árbol de la figura



se representa por

```

ejÁrbol_1 = Nodo 4 (Nodo 2 (Nodo 1 Hoja Hoja)
                        (Nodo 3 Hoja Hoja))
                (Nodo 6 (Nodo 5 Hoja Hoja)
                        (Nodo 7 Hoja Hoja))
  
```

### Definición de funciones sobre árboles

- Las funciones sobre árboles se pueden definir mediante análisis de patrones con las funciones constructoras.
- `tamaño a` es el tamaño del árbol `a`; es decir, el número de nodos internos. Por ejemplo,

```
| tamaño ejÁrbol_1 ~> 7
```

```

tamaño :: Árbol a -> Int
tamaño Hoja           = 0
tamaño (Nodo x a1 a2) = 1 + tamaño a1 + tamaño a2
  
```

## Otras formas de árboles

- Árboles cuyos elementos no están en los nodos (como en `Árbol`), sino que están solamente en las hojas:

```
data Árbol2 a = Nodo2 (Árbol2 a) (Árbol2 a)
              | Fin2 a
```

- Árboles con información del tipo `a` en los nodos, e información del tipo `b` en las hojas:

```
data Árbol3 a b = Nodo3 a (Árbol3 a b) (Árbol3 a b)
                 | Fin3 b
```

- Árboles que se dividen en tres en los nodos, y no en dos:

```
data Árbol4 a = Nodo4 a (Árbol4 a) (Árbol4 a) (Árbol4 a)
               | Fin4
```

- Árboles cuyo número de ramas bifurcadas de un nodo son variables:

```
data Árbol5 a = Nodo5 a [Árbol5 a]
```

- Árboles cuyos nodos solamente tienen una rama bifurcada:

```
data Árbol6 a = Nodo6 a (Árbol6 a)
               | Fin6
```

- Árboles con diferentes tipos de nodos:

```
data Árbol7 a b = Nodo7a Int a (Árbol7 a b) (Árbol7 a b)
                 | Nodo7b Char (Árbol7 a b)
                 | Fin7a b
                 | Fin7b Int
```

## 4.4. Árboles de búsqueda

### Búsqueda en listas

- Búsqueda en lista: `elem e l` se verifica si `e` es un elemento de `l`.
- Búsqueda en lista ordenada: `elem_ord e l` se verifica si `e` es un elemento de la lista ordenada `l`. Por ejemplo,

```
elem_ord 3 [1,3,5] ~> True
elem_ord 2 [1,3,5] ~> False
```

```
elem_ord :: Ord a => a -> [a] -> Bool
elem_ord _ [] = False
elem_ord e (x:xs) | x < e = elem_ord e xs
                  | x == e = True
                  | otherwise = False
```

## Árbol de búsqueda

- Un **árbol de búsqueda** es un árbol binario en el que todos los valores en el subárbol izquierdo son menores que el valor en el nodo mismo, y que todos los valores en el subárbol derecho son mayores. Por ejemplo, el ejÁrbol\_1 es un árbol de búsqueda.
- elemÁrbol e x se verifica si e es un elemento del árbol de búsqueda x. Por ejemplo,

```
elemÁrbol 5 ejÁrbol_1 ~> True
elemÁrbol 9 ejÁrbol_1 ~> False
```

```
elemÁrbol :: Ord a => a -> Árbol a -> Bool
elemÁrbol e Hoja = False
elemÁrbol e (Nodo x izq der) | e==x = True
                             | e<x = elemÁrbol e izq
                             | e>x = elemÁrbol e der
```

## Construcción de un árbol de búsqueda

- insertaÁrbol e ab inserta el elemento e en el árbol de búsqueda ab. Por ejemplo,

```
Main> insertaÁrbol 3 ejÁrbol_1
Nodo 4 (Nodo 2 (Nodo 1 Hoja Hoja)
        (Nodo 3 (Nodo 3 Hoja Hoja) Hoja))
      (Nodo 6 (Nodo 5 Hoja Hoja) (Nodo 7 Hoja Hoja))
```

```
insertaÁrbol :: Ord a => a -> Árbol a -> Árbol a
insertaÁrbol e Hoja = Nodo e Hoja Hoja
insertaÁrbol e (Nodo x izq der)
  | e <= x = Nodo x (insertaÁrbol e izq) der
  | e > x = Nodo x izq (insertaÁrbol e der)
```



- `listaÁrbol 1` es el árbol de búsqueda obtenido a partir de la lista 1. Por ejemplo,

```
Main> listaÁrbol [3,2,4,1]
Nodo 1
  Hoja
  (Nodo 4
    (Nodo 2
      Hoja
      (Nodo 3 Hoja Hoja))
    Hoja)
```

```
listaÁrbol :: Ord a => [a] -> Árbol a
listaÁrbol = foldr insertaÁrbol Hoja
```

### Ordenación mediante árboles de búsqueda

- `aplana ab` es la lista obtenida aplanando el árbol `ab`. Por ejemplo,

```
aplana (listaÁrbol [3,2,4,1]) ~> [1,2,3,4]
```

```
aplana :: Árbol a -> [a]
aplana Hoja = []
aplana (Nodo x izq der) =
  aplana izq ++ [x] ++ aplana der
```

- `ordenada_por_árbol 1` es la lista 1 ordenada mediante árbol de búsqueda. Por ejemplo,

```
ordenada_por_árbol [1,4,3,7,2] ~> [1,2,3,4,7]
```

```
ordenada_por_árbol :: Ord a => [a] -> [a]
ordenada_por_árbol = aplana . listaÁrbol
```

## 4.5. Usos especiales de definiciones de datos

### Tipos finitos

- Un **tipo finito** es un tipo que contiene exactamente tantos elementos como funciones constructoras.
- Ejemplo de tipo finito:

```
data Dirección = Norte | Sur | Este | Oeste
```

- mueve d p es el punto obtenido moviendo el punto p una unidad en la dirección d. Por ejemplo,

```
|mueva Sur (mueve Este (4,7)) ~> (5,6)
```

```
mueva :: Dirección -> (Int,Int) -> (Int,Int)
mueve Norte (x,y) = (x,y+1)
mueve Sur    (x,y) = (x,y-1)
mueve Este  (x,y) = (x+1,y)
mueve Oeste (x,y) = (x-1,y)
```

### Unión de tipos

- Un entero-o-carácter es un objeto de la forma Ent x, donde x es un entero, o Car y, donde y es un carácter.

```
data Ent_o_Car = Ent Int | Car Char deriving Show
```

```
Main> :set +t
Main> Ent 3
Ent 3 :: Ent_o_Car
Main> Car 'd'
Car 'd' :: Ent_o_Car
Main> [Ent 3, Car 'd', Car 'e', Ent 7]
[Ent 3,Car 'd',Car 'e',Ent 7] :: [Ent_o_Car]
Main> :type Ent
Ent :: Int -> Ent_o_Car
Main> :type Car
Car :: Char -> Ent_o_Car
```

### Tipos abstractos

- Un [tipo abstracto](#) consiste en una definición de datos y una serie de funciones que se pueden aplicar al tipo definido.
- Los racionales como tipo abstracto:

```
data Ratio = Rac Int Int
```

- Los racionales se escriben en forma simplificada:

```
instance Show Ratio where
  show (Rac x 1) = show x
  show (Rac x y) = show x' ++ "/" ++ show y'
                  where (Rac x' y') = reduce (Rac x y)
```

- Ejemplos de números racionales:

```
rCero   = Rac 0 1
rUno    = Rac 1 1
rDos    = Rac 2 1
rTres   = Rac 3 1
rMedio  = Rac 1 2
rTercio = Rac 1 3
rCuarto = Rac 1 4
```

```
Main> :set +t
Main> rDos
2 :: Ratio
Main> rTercio
1/3 :: Ratio
```

- `reduce x` es el número racional `x` simplificado. Por ejemplo,

```
reduce (Rac 12 24) ~> 1/2
reduce (Rac 12 -24) ~> -1/2
reduce (Rac -12 -24) ~> 1/2
reduce (Rac -12 24) ~> -1/2
```

```
reduce :: Ratio -> Ratio
reduce (Rac n d) =
  Rac (((signum d)*n) 'div' m) ((abs d) 'div' m)
  where m = gcd n d
```

- Operaciones con números racionales. Por ejemplo,

```
rMul (Rac 1 2) (Rac 2 3) ~> 1/3
rDiv (Rac 1 2) (Rac 1 4) ~> 2
rSum (Rac 1 2) (Rac 3 4) ~> 5/4
rRes (Rac 1 2) (Rac 3 4) ~> -1/4
```

```
rMul, rDiv, rSum, rRes :: Ratio -> Ratio -> Ratio
rMul (Rac a b) (Rac c d) = reduce (Rac (a*c) (b*d))
rDiv (Rac a b) (Rac c d) = reduce (Rac (a*d) (b*c))
rSum (Rac a b) (Rac c d) = reduce (Rac (a*d+b*c) (b*d))
rRes (Rac a b) (Rac c d) = reduce (Rac (a*d-b*c) (b*d))
```

## Bibliografía

1. H. C. Cunningham (2007) *Notes on Functional Programming with Haskell*.
2. J. Fokker (1996) *Programación funcional*.
3. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J. Gallardo (2004). *Razonando con Haskell (Un curso sobre programación funcional)*.
4. S. Thompson (1999) *Haskell: The Craft of Functional Programming*.
5. E.P. Wentworth (1994) *Introduction to Funcional Programming*.

## **Capítulo 4**

# **Aplicaciones de programación funcional**

# Programación declarativa (2008–09)

Tema 4: Aplicaciones de programación funcional

José A. Alonso Jiménez

## Índice

<b>1. Funciones combinatorias</b>	<b>1</b>
1.1. Segmentos y sublistas . . . . .	1
1.2. Permutaciones y combinaciones . . . . .	2
1.3. El patrón alias @ . . . . .	4
<b>2. Rompecabezas lógicos</b>	<b>5</b>
2.1. El problema de las reinas . . . . .	5
2.2. Números de Hamming . . . . .	6
<b>3. Búsqueda en grafos</b>	<b>7</b>
3.1. Búsqueda en profundidad grafos . . . . .	7

# 1. Funciones combinatorias

## 1.1. Segmentos y sublistas

### Segmentos iniciales

- `iniciales l` es la lista de los segmentos iniciales de la lista `l`. Por ejemplo,

```
iniciales [2,3,4]  ~> [[], [2], [2,3], [2,3,4]]
iniciales [1,2,3,4] ~> [[], [1], [1,2], [1,2,3], [1,2,3,4]]
```

```
iniciales :: [a] -> [[a]]
iniciales []      = [[]]
iniciales (x:xs) = [] : [x:ys | ys <- iniciales xs]
```

### Segmentos finales

- `finales l` es la lista de los segmentos finales de la lista `l`. Por ejemplo,

```
finales [2,3,4]  ~> [[2,3,4], [3,4], [4], []]
finales [1,2,3,4] ~> [[1,2,3,4], [2,3,4], [3,4], [4], []]
```

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

### Segmentos

- `segmentos l` es la lista de los segmentos de la lista `l`. Por ejemplo,

```
Main> segmentos [2,3,4]
[[], [4], [3], [3,4], [2], [2,3], [2,3,4]]
Main> segmentos [1,2,3,4]
[[], [4], [3], [3,4], [2], [2,3], [2,3,4],
 [1], [1,2], [1,2,3], [1,2,3,4]]
```

```
segmentos :: [a] -> [[a]]
segmentos []      = [[]]
segmentos (x:xs) =
  segmentos xs ++ [x:ys | ys <- iniciales xs]
```

## Sublistas

- `sublistas l` es la lista de las sublistas de la lista `l`. Por ejemplo,

```
Main> sublistas [2,3,4]
[[2,3,4],[2,3],[2,4],[2],[3,4],[3],[4],[]]
Main> sublistas [1,2,3,4]
[[1,2,3,4],[1,2,3],[1,2,4],[1,2],[1,3,4],[1,3],[1,4],[1],
 [2,3,4],[2,3],[2,4],[2],[3,4],[3],[4],[]]
```

```
sublistas :: [a] -> [[a]]
sublistas [] = [[]]
sublistas (x:xs) = [x:ys | ys <- sub] ++ sub
                  where sub = sublistas xs
```

## 1.2. Permutaciones y combinaciones

### Permutaciones

- `permutaciones l` es la lista de las permutaciones de la lista `l`. Por ejemplo,

```
Main> permutaciones [2,3]
[[2,3],[3,2]]
Main> permutaciones [1,2,3]
[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
```

- Definición recursiva:

```
permutaciones :: Eq a => [a] -> [[a]]
permutaciones [] = [[]]
permutaciones xs =
  [a:p | a <- xs, p <- permutaciones(xs \\ [a])]
```

- `xs \\ ys` es la lista de los elementos de `xs` que no pertenecen a `ys`. Para usarla, hay que importarla: `import Data.List ((\\))`

- Definición alternativa:

```
permutaciones' :: [a] -> [[a]]
permutaciones' [] = [[]]
permutaciones' (x:xs) = [zs | ys <- permutaciones' xs,
                          zs <- intercala x ys]
```



donde `intercala x l` es la lista de las listas obtenidas intercalando `x` entre los elementos de la lista `l`. Por ejemplo,

```
|intercala 1 [2,3] ~> [[1,2,3],[2,1,3],[2,3,1]]
```

```
intercala :: a -> [a] -> [[a]]
intercala e []      = [[e]]
intercala e (x:xs) =
  (e:x:xs) : [(x:ys) | ys <- (intercala e xs)]
```

- La segunda definición es más eficiente. En efecto,

```
Main> :set +s
Main> permutaciones [1,2,3]
[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
(429 reductions, 812 cells)
Main> permutaciones' [1,2,3]
[[1,2,3],[2,1,3],[2,3,1],[1,3,2],[3,1,2],[3,2,1]]
(267 reductions, 485 cells)
```

## Combinaciones

- `combinaciones n l` es la lista de las combinaciones  $n$ -arias de la lista `l`. Por ejemplo,

```
Main> combinaciones 2 [1,2,3,4]
[[1,2],[1,3],[1,4],[2,3],[2,4],[3,4]]
```

- Definición mediante sublistas:

```
combinaciones :: Int -> [a] -> [[a]]
combinaciones n xs =
  [ys | ys <- sublistas xs, length ys == n]
```

- Definición directa:

```
combinaciones' :: Int -> [a] -> [[a]]
combinaciones' 0 _      = [[]]
combinaciones' _ []    = []
combinaciones' (n+1) (x:xs) =
  [x:ys | ys <- combinaciones' n xs] ++
  combinaciones' (n+1) xs
```

- Comparación de eficiencia:

```

Main> combinaciones 1 [1..10]
[[1],[2],[3],[4],[5],[6],[7],[8],[9],[10]]
(42363 reductions, 52805 cells)
Main> combinaciones' 1 [1..10]
[[1],[2],[3],[4],[5],[6],[7],[8],[9],[10]]
(775 reductions, 1143 cells)

```

### 1.3. El patrón alias @

#### El patrón alias @

- Definición de finales con patrones:

```

finales_1 []      = [[]]
finales_1 (x:xs) = (x:xs) : finales_1 xs

```

- Definición de finales con selectores:

```

finales_2 [] = [[]]
finales_2 xs = xs : finales_2 (tail xs)

```

- Definición de finales con alias::

```

finales_3 []      = [[]]
finales_3 l@(x:xs) = l : finales_3 xs

```

- El patrón `l@(x:xs)` se lee “l como x:xs”.

#### Definición con alias

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

```

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

```

```

----- Prelude -----
dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile p []          = []
dropWhile p ys@(x:xs)
  | p x                 = dropWhile p xs
  | otherwise           = ys

```

## 2. Rompecabezas lógicos

### 2.1. 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 `[1ex]`

	R		
			R
R			
		R	

`[1ex]`

```
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)
```

- Esta solución está basada en [Wentworth 94] p. 95.

## 2.2. Números de Hamming

### Números de Hamming

- Enunciado: Los números de Hamming forman una sucesión estrictamente creciente de números que cumplen las siguientes condiciones:

1. El número 1 está en la sucesión.
2. Si  $x$  está en la sucesión, entonces  $2x$ ,  $3x$  y  $5x$  también están.
3. 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 zs` 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
```

## 3. Búsqueda en grafos

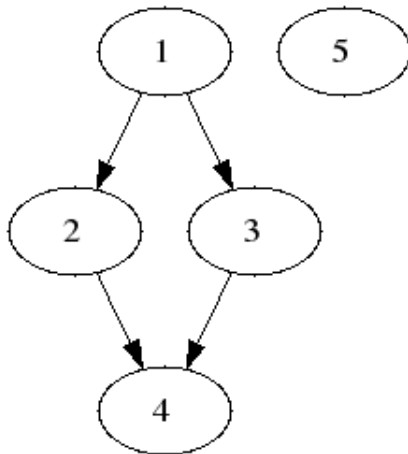
### 3.1. Búsqueda en profundidad grafos

#### Representación de grafos

- Un grafo de tipo `v` es un tipo de datos compuesto por la lista de vértices y la función que asigna a cada vértice la lista de sus sucesores.

```
data Grafo v = G [v] (v -> [v])
```

- `ej_grafo` es la representación del grafo



```
ej_grafo = G [1..5] suc
  where suc 1 = [2,3]
         suc 2 = [4]
         suc 3 = [4]
         suc _ = []
```

#### Búsqueda de los caminos en un grafo

- `caminosDesde g o te vis` es la lista de los caminos en el grafo `g` desde el vértice origen `o` hasta vértices finales (i.e los que verifican el test de encontrado `te`) sin volver a pasar por los vértices visitados `vis`. Por ejemplo,

```
|caminosDesde ej_grafo 1 (==4) [] ~> [[4,2,1],[4,3,1]]
```

```
caminosDesde :: Eq a => Grafo a -> a -> (a -> Bool) -> [a] -> [[a]]
caminosDesde g o te vis
```

```

| te o      = [o:vis]
| otherwise = concat [caminosDesde g o' te (o:vis)
                      | o' <- suc o,
                      notElem o' vis]

where G _ suc = g

```

- camino g u v es un camino (i.e una lista de vértices tales que cada uno es un sucesor del anterior) en el grafo g desde el vértice u al v. Por ejemplo,

```
| camino ej_grafo 1 4 ~> [4,2,1]
```

```

camino :: Eq a => Grafo a -> a -> a -> [a]
camino g u v = head (caminosDesde g u (== v) [])

```

1. H. C. Cunningham (2007) *Notes on Functional Programming with Haskell*.
2. J. Fokker (1996) *Programación funcional*.
3. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J. Gallardo (2004). *Razonando con Haskell (Un curso sobre programación funcional)*.
4. S. Thompson (1999) *Haskell: The Craft of Functional Programming*.
5. E.P. Wentworth (1994) *Introduction to Funcional Programming*.

## **Capítulo 5**

# **Razonamiento sobre programas**

# Programación declarativa (2008–09)

## Tema 5: Razonamiento sobre programas

José A. Alonso Jiménez

### Índice

<b>1. Razonamiento ecuacional</b>	<b>1</b>
1.1. Cálculo con longitud . . . . .	1
1.2. Propiedad de intercambia . . . . .	1
<b>2. Razonamiento por inducción sobre listas</b>	<b>1</b>
2.1. Esquema de inducción sobre listas . . . . .	1
2.2. Asociatividad de ++ . . . . .	2
2.3. [] es la identidad para ++ por la derecha . . . . .	3
2.4. Relación entre length y ++ . . . . .	3
2.5. Relación entre take y drop . . . . .	5
2.6. La concatenación de listas vacías es vacía . . . . .	6
2.7. Relación entre sum y map . . . . .	7
<b>3. Equivalencia de funciones</b>	<b>8</b>



## 1. Razonamiento ecuacional

### 1.1. Cálculo con longitud

#### Cálculo con longitud

- Programa:

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

- Propiedad:  $\text{longitud } [2,3,1] = 3$

- 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
```

### 1.2. Propiedad de intercambia

#### Propiedad de intercambia

- Programa:

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

- Propiedad:

$$(\forall x :: a)(\forall y :: b)\text{intercambia}(\text{intercambia}(x,y)) = (x,y).$$

- Demostración:

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

## 2. Razonamiento por inducción sobre listas

### 2.1. Esquema de inducción sobre listas

#### 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  $P(y:ys)$ .

## 2.2. Asociatividad de ++

### Asociatividad de ++

- Programa:

```

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

- Propiedad:  $(\forall xs, ys, zs :: [a]) xs ++ (ys ++ zs) = (xs ++ ys) ++ zs$
- Comprobación con QuickCheck:

```

prop_asociatividad_conc :: [Int] -> [Int] -> [Int] -> Bool
prop_asociatividad_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
 
$$(a : as) ++ (ys ++ zs) = ((a : as) ++ ys) ++ zs$$

$$(a : as) ++ (ys ++ zs)$$

$$= a : (as ++ (ys ++ zs)) \quad [\text{por ++.2}]$$

$$= a : ((as ++ ys) ++ zs) \quad [\text{por hip. de inducción}]$$

$$= (a : (as ++ ys)) ++ zs \quad [\text{por ++.2}]$$

$$= ((a : as) ++ ys) ++ zs \quad [\text{por ++.2}]$$

## 2.3. `[]` es la identidad para `++` por la derecha

### `[]` es la identidad para `++` por la derecha

- Propiedad:  $(\forall xs :: [a]) xs ++ [] = xs$
- Comprobación con QuickCheck:

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

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

- Demostración por inducción en  $xs$ :
  - Caso base  $xs = []$ :
 
$$= [] ++ []$$

$$= [] \quad [\text{por ++.1}]$$
  - Caso inductivo  $xs = (a : as)$ : Suponiendo la hipótesis de inducción  $as ++ [] = as$  hay que demostrar que
 
$$(a : as) ++ [] = (a : as)$$

$$(a : as) ++ []$$

$$= a : (as ++ []) \quad [\text{por ++.2}]$$

$$= a : as \quad [\text{por hip. de inducción}]$$

## 2.4. Relación entre `length` y `++`

### 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:  $(\forall xs, ys :: [a]) \text{length}(xs ++ ys) = (\text{length } xs) + (\text{length } ys)$
- 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  $xs = []$ :
 

$\text{length}([] ++ ys)$	
$= \text{length } ys$	[por ++.1]
$= 0 + (\text{length } ys)$	[por aritmética]
$= (\text{length } []) + (\text{length } ys)$	[por length.1]

- Demostración por inducción en  $xs$ :

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

$\text{length}(as ++ ys) = (\text{length } as) + (\text{length } ys)$	
hay que demostrar que	
$\text{length}((a : as) ++ ys) = (\text{length } (a : as)) + (\text{length } ys)$	
$\text{length}((a : as) ++ ys)$	
$= \text{length}(a : (as ++ ys))$	[por ++.2]
$= 1 + \text{length}(as ++ ys)$	[por length.2]
$= 1 + ((\text{length } as) + (\text{length } ys))$	[por hip. de inducción]
$= (1 + (\text{length } as)) + (\text{length } ys)$	[por aritmética]
$= (\text{length } (a : as)) + (\text{length } ys)$	[por length.2]

## 2.5. Relación entre take y drop

### 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:  $(\forall n :: Nat, xs :: [a]) \text{take } n \text{ xs} ++ \text{drop } n \text{ xs} = xs$

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

Lo demostraremos por inducción en xs:

- Caso base  $xs=[]$ :
 

```

take (m+1) [] ++ drop (m+1) []
= [] ++ []                                [por take.2 y drop.2]
= []                                       [por ++.1]
      
```
- Caso inductivo  $xs=(a:as)$ : Suponiendo la hipótesis de inducción 2
 

```

take (m+1) as ++ drop (m+1) as = as
      
```

 hay que demostrar que
 

```

take (m+1) (a:as) ++ drop (m+1) (a:as) = (a:as)
take (m+1) (a:as) ++ drop (m+1) (a:as)
= (a:(take m as)) ++ (drop m as)         [por take.3 y drop.3]
= (a:((take m as) ++ (drop m as)))      [por ++.2]
= a:as                                   [por hip. de ind. 1]
      
```

## 2.6. La concatenación de listas vacías es vacía

### 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:  $\forall (xs :: [a]) \text{null } xs = \text{null } (xs ++ xs)$ .

- 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,  $\text{null } xs = \text{null } (xs ++ xs)$ .

- 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)`.

## 2.7. Relación entre `sum` y `map`

### Relación entre `sum` y `map`

- Programas:

```

----- Prelude -----
sum :: [Int] -> Int
sum []      = 0                -- sum.1
sum (x:xs) = x + sum xs       -- sum.2

map :: (a -> b) -> [a] -> [b]
map f []    = []              -- map.1
map f (x:xs) = f x : map f xs -- map.2

```

- Propiedad:  $\forall xs :: [Int] . \text{sum} (\text{map} (2*) xs) = 2 * \text{sum} xs$
- Comprobación con QuickCheck:

```

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

```

- Demostración por inducción en `xs`:
  - Caso `[]`: Reduciendo el lado izquierdo

```

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

```

y reduciendo el lado derecho

```

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

```

Luego,  $\text{sum (map (2*) xs)} = 2 * \text{sum xs}$

■ Demostración por inducción en  $xs$ :

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

```

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

```

### 3. Equivalencia de funciones

#### 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:  $(\forall xs :: [a]) \text{inversa1 xs} = \text{inversa2 xs}$
- Comprobación con QuickCheck:



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

- Demostración: Es consecuencia del siguiente lema:

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

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. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J. Gallardo (2004). *Razonando con Haskell (Un curso sobre programación funcional)*.
4. S. Thompson (1999) *Haskell: The Craft of Functional Programming*.
5. E.P. Wentworth (1994) *Introduction to Funcional Programming*.



**Parte II**  
**Programación lógica**



## **Capítulo 6**

### **Introducción a Prolog**

# Programación declarativa (2008–09)

Tema 6: Introducción a Prolog

José A. Alonso Jiménez

## Índice

<b>1. Introducción a la programación lógica</b>	<b>1</b>
1.1. Objetivos de la programación lógica . . . . .	1
1.2. Declarativo vs. imperativo . . . . .	1
1.3. Historia de la programación lógica . . . . .	2
<b>2. Deducción Prolog</b>	<b>3</b>
2.1. Deducción Prolog en lógica proposicional . . . . .	3
2.2. Deducción Prolog en lógica relacional . . . . .	5
2.3. Deducción Prolog en lógica funcional . . . . .	7

## 1. Introducción a la programación lógica

### 1.1. Objetivos de la programación lógica

#### Objetivos de la programación lógica

- Lógica como sistema de especificación y lenguaje de programación.
- Principios:
  - Programas = Teorías.
  - Ejecución = Búsqueda de pruebas.
  - Programación = Modelización.
- Prolog = Programming in Logic.
- Relaciones con otros campos:
  - Inteligencia artificial.
  - Sistemas basados en el conocimiento.

- Procesamiento del lenguaje natural.
- Pensar declarativamente.

## 1.2. Declarativo vs. imperativo

### Declarativo vs. imperativo

- Paradigmas:
  - Imperativo: Se describe *cómo* resolver el problema.
  - Declarativo: Se describe *qué* es el problema.
- Programas:
  - Imperativo: Una sucesión de instrucciones.
  - Declarativo: Un conjunto de sentencias.
- Lenguajes:
  - Imperativo: Pascal, C, Fortran.
  - Declarativo: Prolog, Lisp puro, ML, Haskell, DLV, Smodels.
- Ventajas:
  - Imperativo: Programas rápidos y especializados.
  - Declarativo: Programas generales, cortos y legibles.

## 1.3. Historia de la programación lógica

### Historia de la programación lógica

- 1960: Demostración automática de teoremas.
- 1965: Resolución y unificación (Robinson).
- 1969: QA3, obtención de respuesta (Green).
- 1972: Implementación de Prolog (Colmerauer).
- 1974: Programación lógica (Kowalski).
- 1977: Prolog de Edimburgo (Warren).
- 1981: Proyecto japonés de Quinta Generación.
- 1986: Programación lógica con restricciones.
- 1995: Estándar ISO de Prolog.

## 2. Deducción Prolog

### 2.1. Deducción Prolog en lógica proposicional

#### Deducción Prolog en lógica proposicional

- Base de conocimiento y objetivo:
  - Base de conocimiento:
    - Regla 1: Si un animal es unguilado y tiene rayas negras, entonces es una cebra.
    - Regla 2: Si un animal rumia y es mamífero, entonces es unguilado.
    - Regla 3: Si un animal es mamífero y tiene pezuñas, entonces es unguilado.
    - Hecho 1: El animal es mamífero.
    - Hecho 2: El animal tiene pezuñas.
    - Hecho 3: El animal tiene rayas negras.
  - Objetivo: Demostrar a partir de la base de conocimientos que el animal es una cebra.
- Programa:

```

es_cebra      :- es_ungulado, tiene_rayas_negras. %R1
es_ungulado  :- rumia, es_mamífero.             %R2
es_ungulado  :- es_mamífero, tiene_pezuñas.    %R3
es_mamífero.                                     %H1
tiene_pezuñas.                                  %H2
tiene_rayas_negras.                             %H3

```

- Sesión:

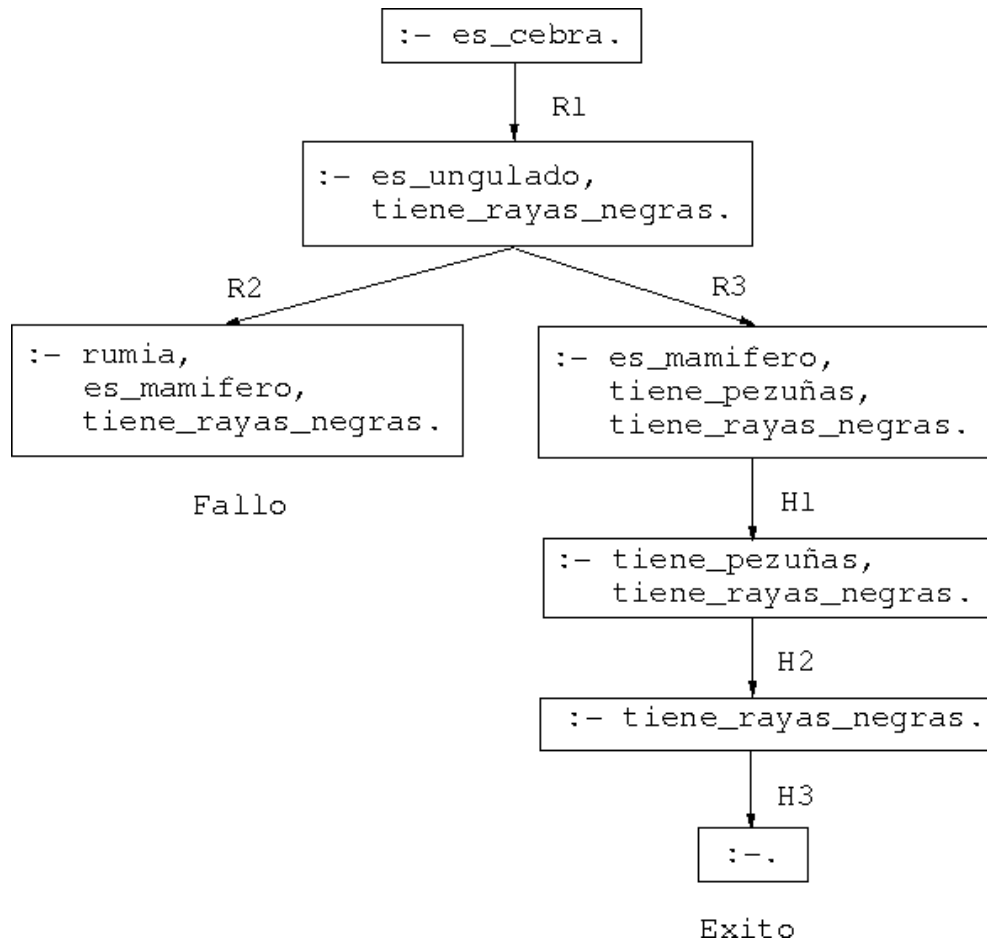
```

> pl
Welcome to SWI-Prolog (Multi-threaded, Version 5.6.20)
Copyright (c) 1990-2006 University of Amsterdam.
?- [animales].
Yes
?- es_cebra.
Yes

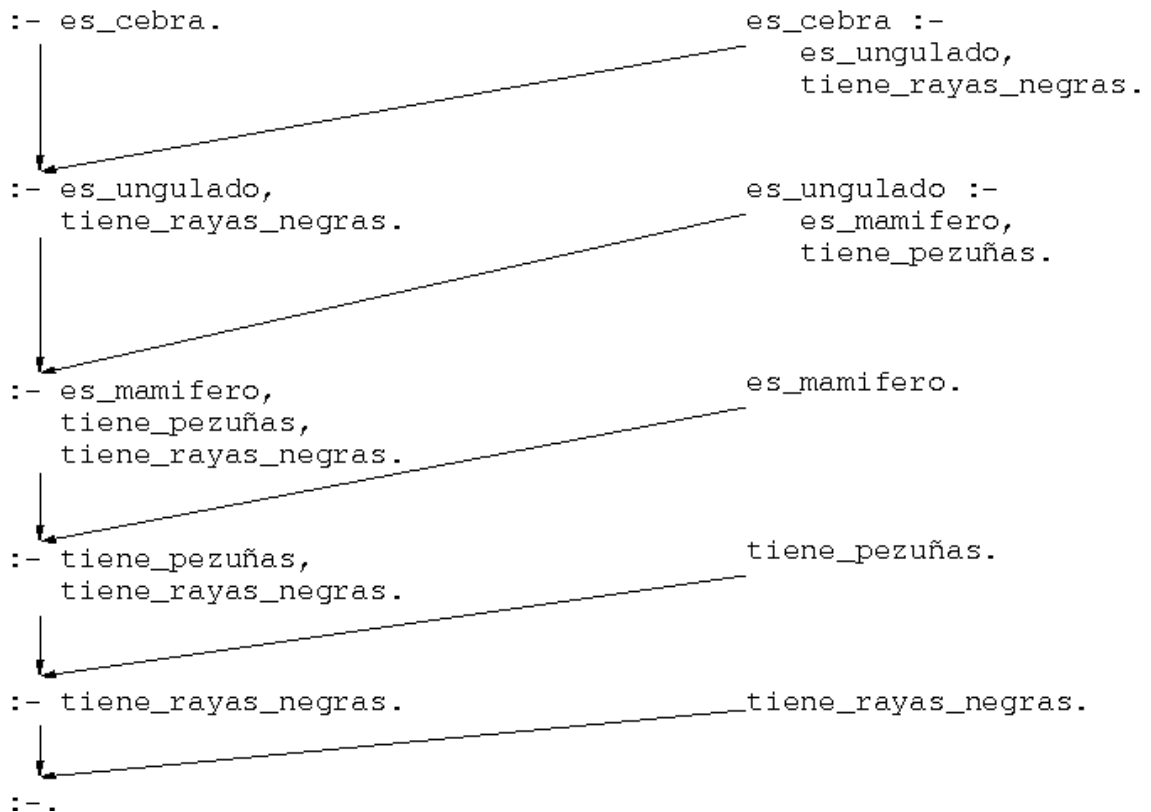
```

- Árbol de deducción:





- Demostración por resolución SLD:



## 2.2. Deducción Prolog en lógica relacional

### Deducción Prolog en lógica relacional

- Base de conocimiento:
  - Hechos 1-4: 6 y 12 son divisibles por 2 y por 3.
  - Hecho 5: 4 es divisible por 2.
  - Regla 1: Los números divisibles por 2 y por 3 son divisibles por 6.

- Programa:

```

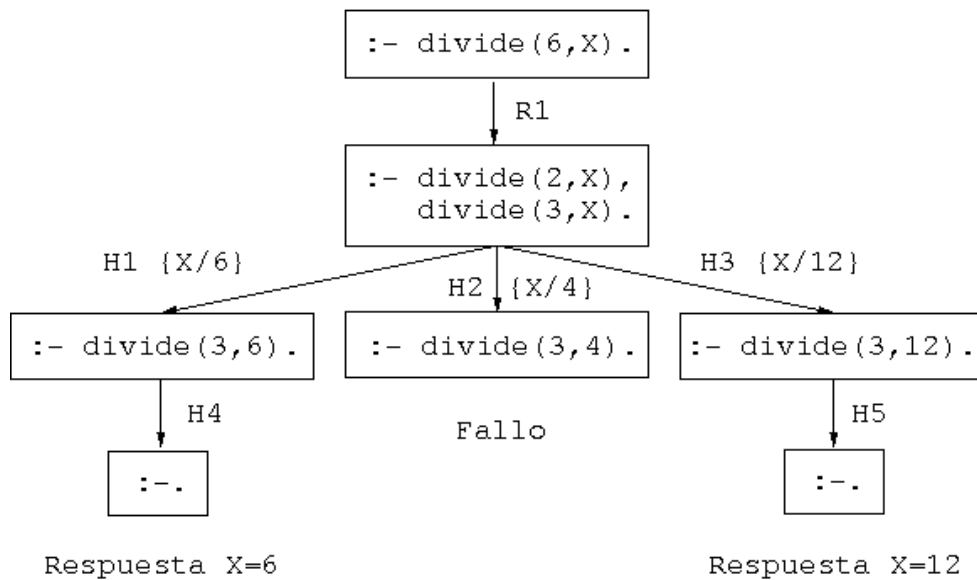
divide(2,6).           % Hecho 1
divide(2,4).           % Hecho 2
divide(2,12).          % Hecho 3
divide(3,6).           % Hecho 4
divide(3,12).          % Hecho 5
divide(6,X) :- divide(2,X), divide(3,X). % Regla 1
  
```

- Símbolos:

- Constantes: 2, 3, 4, 6, 12
- Relación binaria: divide
- Variable: X
- Interpretaciones de la Regla 1:
  - `divide(6,X) :- divide(2,X), divide(3,X).`
  - Interpretación declarativa:  
 $(\forall X)[\text{divide}(2, X) \wedge \text{divide}(3, X) \rightarrow \text{divide}(6, X)]$
  - Interpretación procedimental.
- Consulta: ¿Cuáles son los múltiplos de 6?

```
?- divide(6,X).
X = 6 ;
X = 12 ;
No
```

- Árbol de deducción:



- Comentarios:
  - Unificación.
  - Cálculo de respuestas.
  - Respuestas múltiples.

## 2.3. Deducción Prolog en lógica funcional

### Deducción Prolog en lógica funcional

- Representación de los números naturales:

$0, s(0), s(s(0)), \dots$

- Definición de la suma:

```
0 + Y = Y
s(X) + Y = s(X+Y)
```

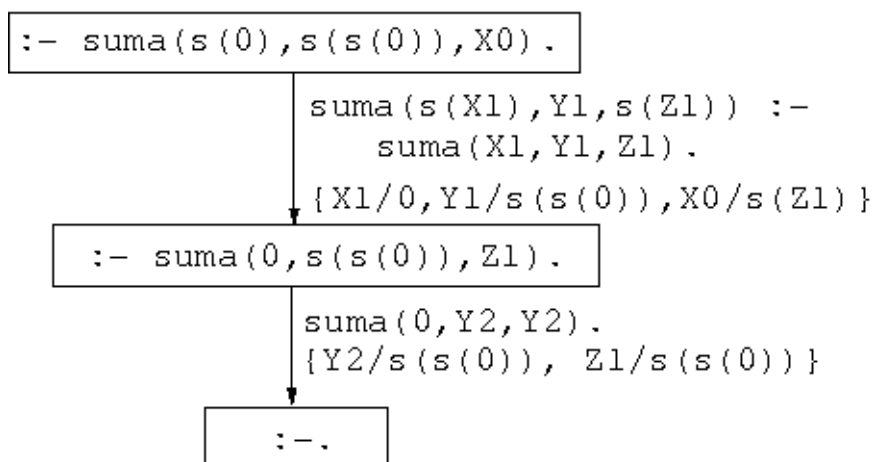
- Programa

```
suma(0, Y, Y).                % R1
suma(s(X), Y, s(Z)) :- suma(X, Y, Z). % R2
```

- Consulta: ¿Cuál es la suma de  $s(0)$  y  $s(s(0))$ ?

```
?- suma(s(0), s(s(0)), X).
X = s(s(s(0)))
Yes
```

- Árbol de deducción:



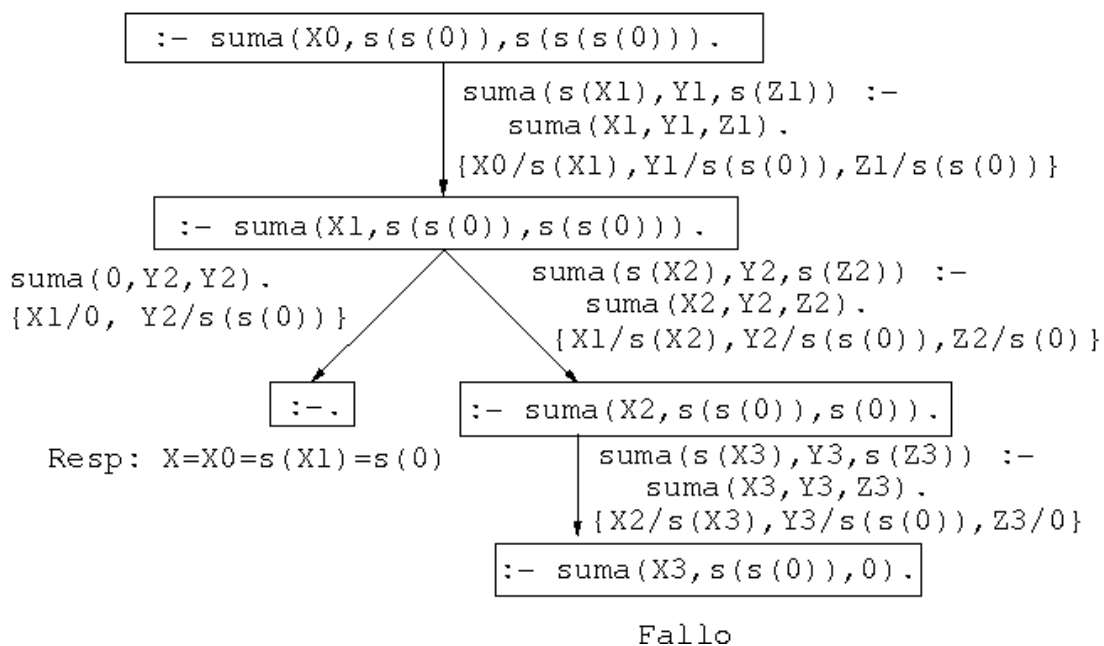
Resp.:  $X = X0 = s(Z1) = s(s(s(0)))$

- Consulta:

- ¿Cuál es la resta de  $s(s(s(0)))$  y  $s(s(0))$ ?
- Sesión:

```
?- suma(X,s(s(0)),s(s(s(0))))).
X = s(0) ;
No
```

■ **Árbol de deducción:**

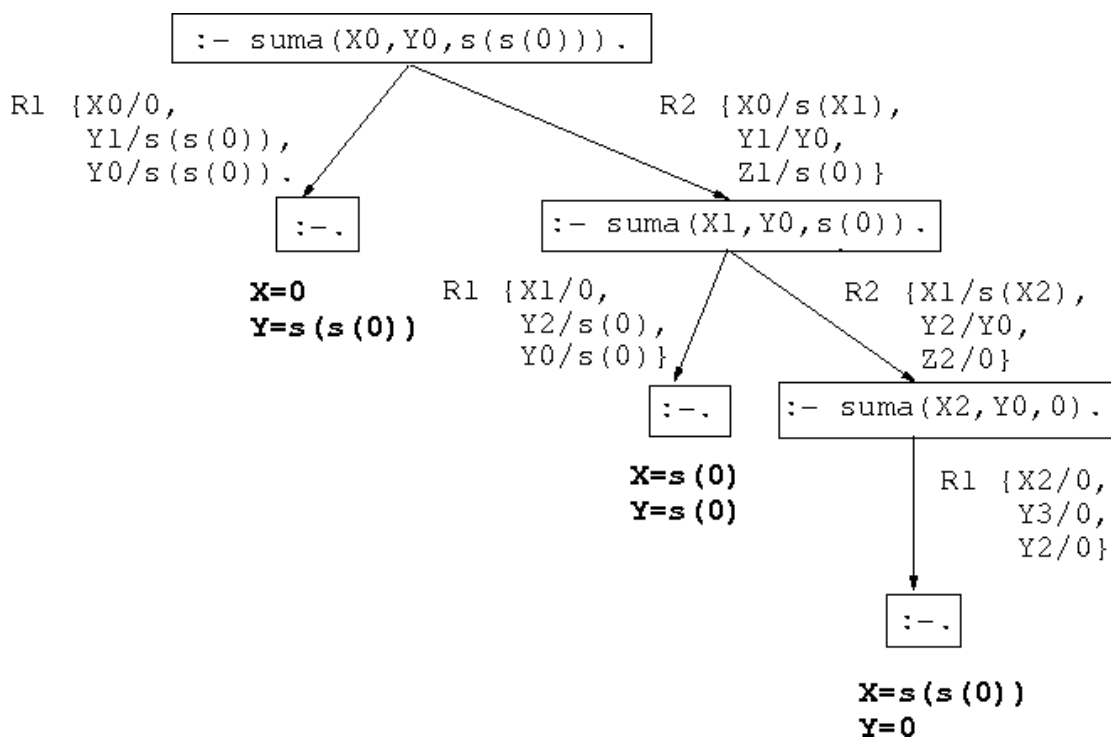


■ **Consulta:**

- **Pregunta:** ¿Cuáles son las soluciones de la ecuación  $X + Y = s(s(0))$ ?
- **Sesión:**

```
?- suma(X,Y,s(s(0))).
X = 0           Y = s(s(0)) ;
X = s(0)       Y = s(0) ;
X = s(s(0))    Y = 0 ;
No
```

■ **Árbol de deducción:**



■ Consulta:

- Pregunta: resolver el sistema de ecuaciones

$$1 + X = Y$$

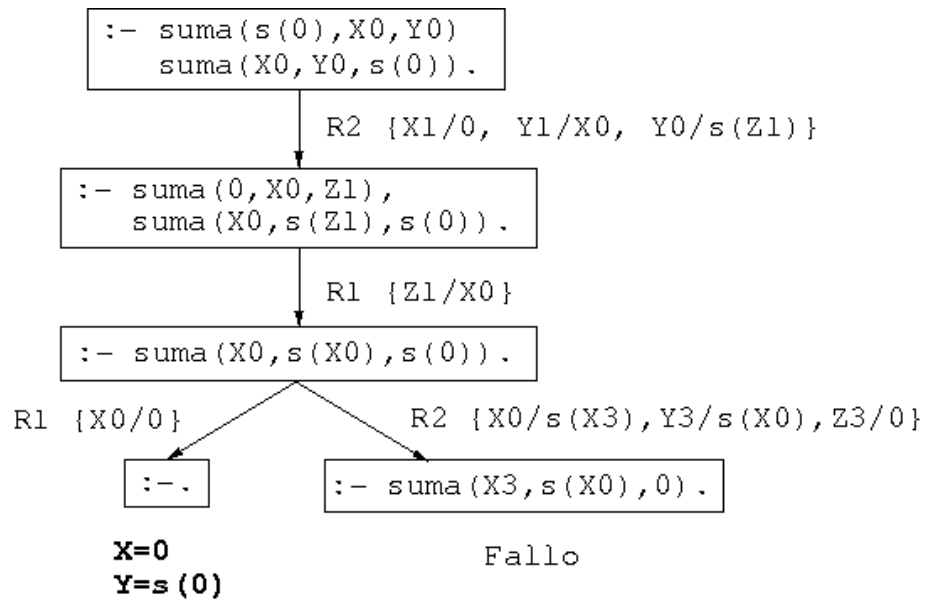
$$X + Y = 1$$

- Sesión:

```

?- suma(s(0), X, Y), suma(X, Y, s(0)).
X = 0
Y = s(0) ;
No
  
```

■ Árbol de deducción:



## Bibliografía

1. J.A. Alonso (2006) *Introducción a la programación lógica con Prolog*.
  - Cap. 0: “Introducción”.
2. I. Bratko (1990) *Prolog Programming for Artificial Intelligence (2nd ed.)*.
  - Cap. 1: “An overview of Prolog”.
  - Cap. 2: “Syntax and meaning of Prolog programs”.
3. W.F. Clocksin y C.S. Mellish (1994) *Programming in Prolog (Fourth Edition)*.
  - Cap. 1: “Tutorial introduction”.
  - Cap. 2: “A closer look”.





# Capítulo 7

## Listas, operadores y aritmética

# Programación declarativa (2008–09)

Tema 7: Listas, operadores y aritmética

José A. Alonso Jiménez

## Índice

<b>1. Listas</b>	<b>1</b>
1.1. Construcción de listas . . . . .	1
1.2. Definición de relaciones sobre listas . . . . .	2
1.2.1. Concatenación de listas . . . . .	2
1.2.2. Relación de pertenencia . . . . .	3
<b>2. Disyunciones</b>	<b>4</b>
<b>3. Operadores</b>	<b>5</b>
3.1. Operadores aritméticos . . . . .	5
3.2. Definición de operadores . . . . .	6
<b>4. Aritmética</b>	<b>7</b>
4.1. Evaluación de expresiones aritméticas . . . . .	7
4.2. Definición de relaciones aritméticas . . . . .	7

# 1. Listas

## 1.1. Construcción de listas

### Construcción de listas

- Definición de listas:
  - La lista vacía `[]` es una lista.
  - Si `L` es una lista, entonces `.(a,L)` es una lista.

- Ejemplos:

```
?- .(X,Y) = [a] .
X = a
Y = []
?- .(X,Y) = [a,b] .
X = a
Y = [b]
?- .(X,.(Y,Z)) = [a,b] .
X = a
Y = b
Z = []
```

### Escritura abreviada

- Escritura abreviada:

```
[X|Y] = .(X,Y)
```

- Ejemplos con escritura abreviada:

```
?- [X|Y] = [a,b] .
X = a
Y = [b]
?- [X|Y] = [a,b,c,d] .
X = a
Y = [b, c, d]
?- [X,Y|Z] = [a,b,c,d] .
X = a
Y = b
Z = [c, d]
```

## 1.2. Definición de relaciones sobre listas

### 1.2.1. Concatenación de listas

#### Definición de concatenación (append)

- *Especificación:*  $\text{conc}(A, B, C)$  se verifica si  $C$  es la lista obtenida escribiendo los elementos de la lista  $B$  a continuación de los elementos de la lista  $A$ . Por ejemplo,

```
?- conc([a,b],[b,d],C).
C=[a,b,b,d]
```

- *Definición 1:*

```
conc(A,B,C) :- A=[], C=B.
conc(A,B,C) :- A=[X|D], conc(D,B,E), C=[X|E].
```

- *Definición 2:*

```
conc([],B,B).
conc([X|D],B,[X|E]) :- conc(D,B,E).
```

#### Consultas con la relación de concatenación

- Analogía entre la definición de  $\text{conc}$  y la de suma,
- ¿Cuál es el resultado de concatenar las listas  $[a, b]$  y  $[c, d, e]$ ?

```
?- conc([a,b],[c,d,e],L).
L=[a,b,c,d,e]
```

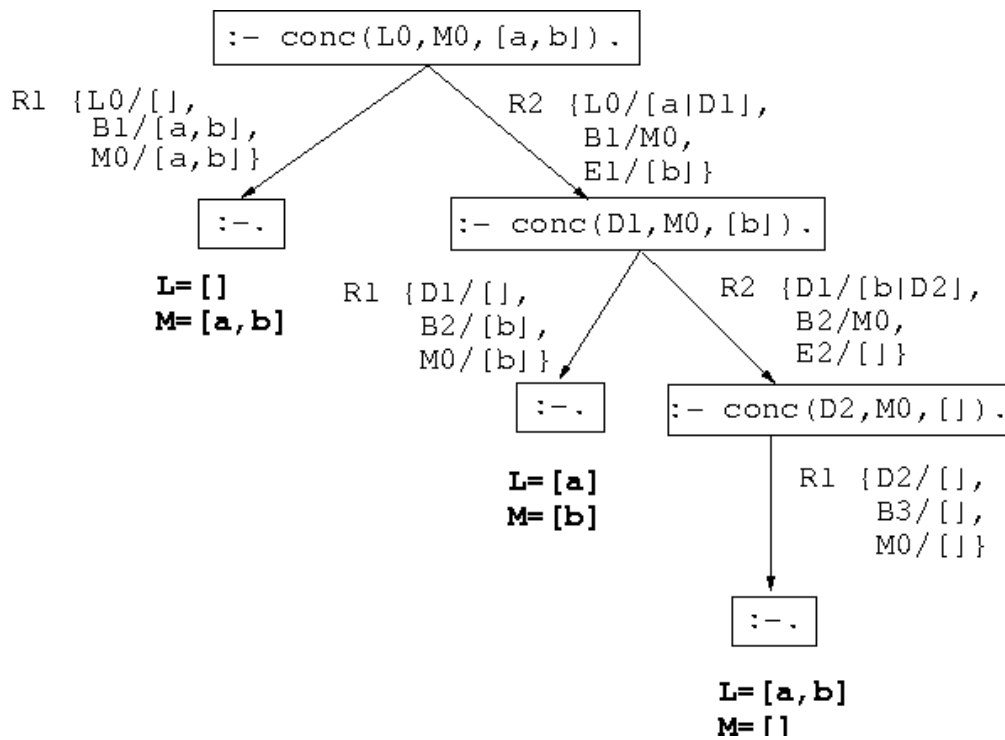
- ¿Qué lista hay que añadirle a la lista  $[a, b]$  para obtener  $[a, b, c, d]$ ?

```
?- conc([a,b],L,[a,b,c,d]).
L=[c,d]
```

- ¿Qué dos listas hay que concatenar para obtener  $[a, b]$ ?

```
?- conc(L,M,[a,b]).
L=[]      M=[a,b] ;
L=[a]    M=[b] ;
L=[a,b]  M=[] ;
No
```

**Árbol de deducción de `?- conc(L,M,[a,b]).`**



**1.2.2. Relación de pertenencia**

**Definición de la relación de pertenencia (member)**

- *Especificación:* pertenece(X,L) se verifica si X es un elemento de la lista L.
- *Definición 1:*

```

pertenece(X,[X|L]).
pertenece(X,[Y|L]) :- pertenece(X,L).
  
```

- *Definición 2:*

```

pertenece(X,[X|_]).
pertenece(X,[_|L]) :- pertenece(X,L).
  
```

**Consultas con la relación de pertenencia**

`?- pertenece(b,[a,b,c]).`

Yes

`?- pertenece(d,[a,b,c]).`

```

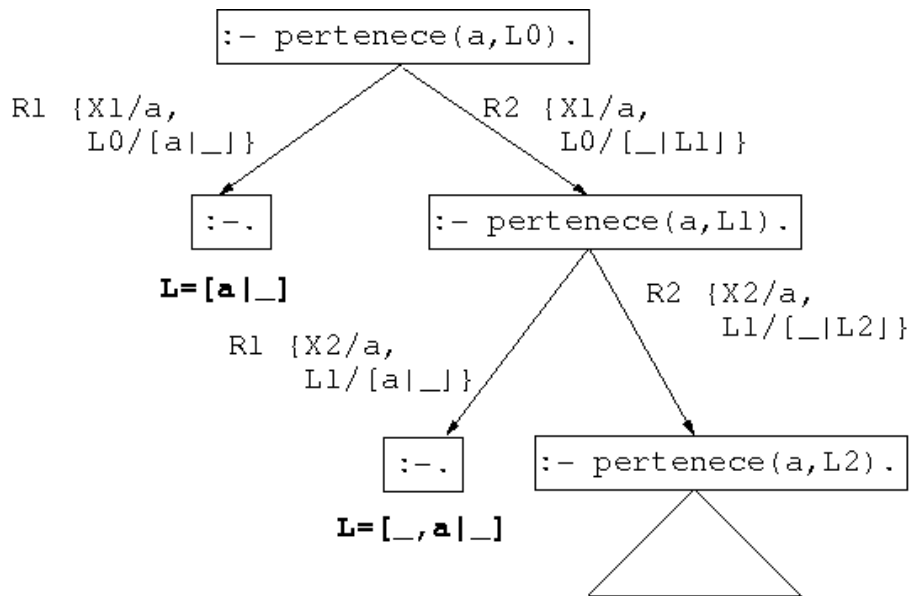
No
?- pertenece(X,[a,b,a]).

X = a ;
X = b ;
X = a ;
No
?- pertenece(a,L).

L = [a|_G233] ;
L = [_G232, a|_G236] ;
L = [_G232, _G235, a|_G239]
Yes

```

Árbol de deducción de `?- pertenece(a,L).`



## 2. Disyunciones

### Disyunciones

- Definición de pertenece con disyunción

```

pertenece(X,[Y|L]) :- X=Y ; pertenece(X,L).

```

- Definición equivalente sin disyunción

```

pertenece(X,[Y|L]) :- X=Y.
pertenece(X,[Y|L]) :- pertenece(X,L).

```

## 3. Operadores

### 3.1. Operadores aritméticos

#### Ejemplos de operadores aritméticos

- Ejemplos de notación infija y prefija en expresiones aritméticas:

|?- +(X,Y) = a+b.

|X = a

|Y = b

|?- +(X,Y) = a+b+c.

|X = a+b

|Y = c

|?- +(X,Y) = a+(b+c).

|X = a

|Y = b+c

|?- a+b+c = (a+b)+c.

|Yes

|?- a+b+c = a+(b+c).

|No

#### Ejemplos de asociatividad y precedencia

- Ejemplos de asociatividad:

|?- X^Y = a^b^c.

|X = a            Y = b^c

|?- a^b^c = a^(b^c).

|Yes

- Ejemplo de precedencia

|?- X+Y = a+b\*c.

|X = a            Y = b\*c

|?- X\*Y = a+b\*c.

```
| No
|- X*Y = (a+b)*c.
```

```
| X = a+b      Y = c
|- a+b*c = (a+b)*c.
```

```
| No
```

### Operadores aritméticos predefinidos

Precedencia	Tipo	Operadores	
500	yfx	+,-	Infijo asocia por la izquierda
500	fx	-	Prefijo no asocia
400	yfx	*, /	Infijo asocia por la izquierda
200	xfy	^	Infijo asocia por la derecha

## 3.2. Definición de operadores

### Definición de operadores

- Definición (ejemplo\_operadores.pl)

```
:-op(800,xfx,estudian).
:-op(400,xfx,y).

juan y ana estudian lógica.
```

- Consultas

```
?- [ejemplo_operadores].
?- Quienes estudian lógica.

Quienes = juan y ana
?- juan y Otro estudian Algo.

Otro = ana
Algo = lógica
```



## 4. Aritmética

### 4.1. Evaluación de expresiones aritméticas

#### Evaluación de expresiones aritméticas

- Evaluación de expresiones aritmética con `is`.

```
|?- X is 2+3^3.
```

```
|X = 29
```

```
|?- X is 2+3, Y is 2*X.
```

```
|X = 5      Y = 10
```

- Relaciones aritméticas: `<`, `=<`, `>`, `>=`, `:=` y `/=`

```
|?- 3 =< 5.
```

```
|Yes
```

```
|?- 3 > X.
```

```
|% [WARNING: Arguments are not sufficiently instantiated]
```

```
|?- 2+5 = 10-3.
```

```
|No
```

```
|?- 2+5 := 10-3.
```

```
|Yes
```

### 4.2. Definición de relaciones aritméticas

#### Definición del factorial

- `factorial(X,Y)` se verifica si Y es el factorial de X. Por ejemplo,

```
|?- factorial(3,Y).
```

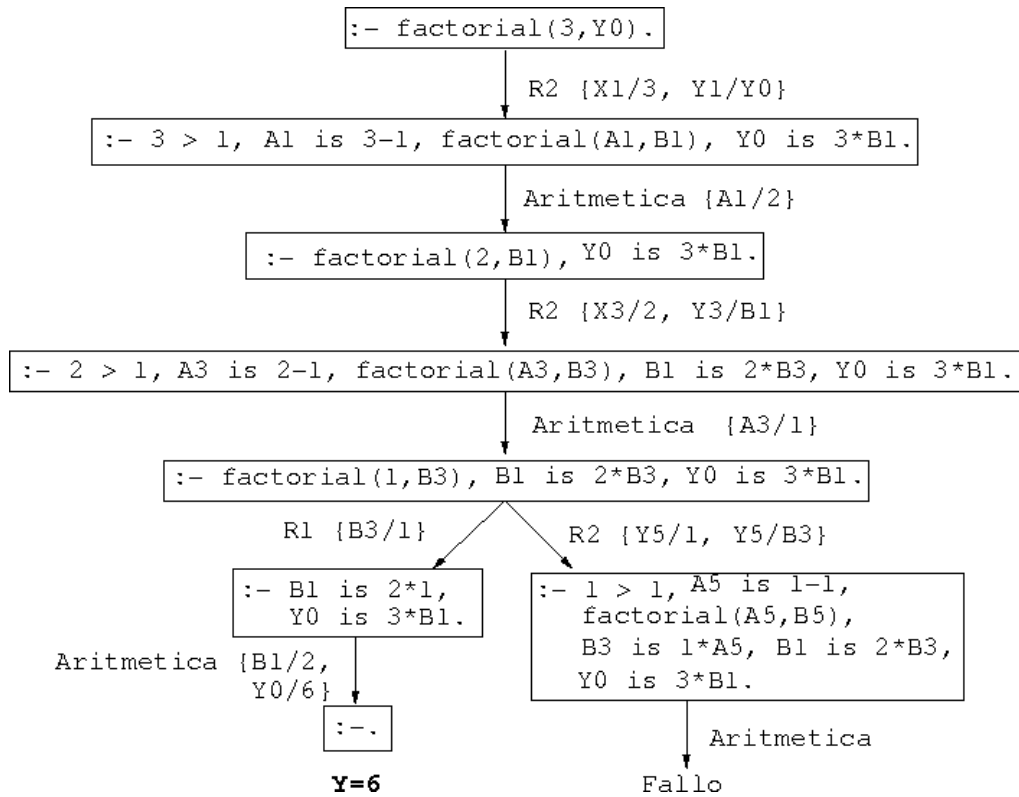
```
|Y = 6 ;
```

```
|No
```

- Definición:

```
factorial(1,1).
factorial(X,Y) :-
    X > 1,
    A is X - 1,
    factorial(A,B),
    Y is X * B.
```

Árbol de deducción de `?- factorial(3,Y).`



## Bibliografía

### Bibliografía

1. J.A. Alonso *Introducción a la programación lógica con Prolog.*
2. Bratko, I. *Prolog Programming for Artificial Intelligence (2nd ed.)* (Addison-Wesley, 1990)
3. Clocksin, W.F. y Mellish, C.S. *Programming in Prolog (Fourth Edition)* (Springer Verlag, 1994)
4. Covington, M.A.; Nute, D. y Vellino, A. *Prolog Programming in Depth* (Prentice Hall, 1997)
5. Sterling, L. y Shapiro, E. *L'art de Prolog* (Masson, 1990)
6. Van Le, T. *Techniques of Prolog Programming* (John Wiley, 1993)

# Capítulo 8

## Estructuras

# Programación declarativa (2008–09)

Tema 8: Estructuras

José A. Alonso Jiménez

## Índice

<b>1. Segmentos horizontales y verticales</b>	<b>1</b>
1.1. Representación de segmentos . . . . .	1
1.2. Consultas sobre segmentos . . . . .	1
<b>2. Base de datos familiar</b>	<b>2</b>
2.1. Representación de la base de datos familiar . . . . .	2
2.2. Consultas a la base de datos familiar . . . . .	4
2.3. Extensiones y consultas a la base de datos familiar . . . . .	4
<b>3. Autómatas no deterministas</b>	<b>5</b>
3.1. Representación de un autómata no determinista . . . . .	5
3.2. Simulación de los autómatas no deterministas . . . . .	7
3.3. Consultas al autómata . . . . .	7
<b>4. Problema de planificación</b>	<b>8</b>
4.1. Representación del problema del mono . . . . .	8
4.2. Solución del problema del mono . . . . .	8

# 1. Segmentos horizontales y verticales

## 1.1. Representación de segmentos

### Definición de segmentos verticales y horizontales

- Representación:
  - punto( $X, Y$ )
  - seg( $P1, P2$ )
- `horizontal(S)` se verifica si el segmento  $S$  es horizontal.
- `vertical(S)` se verifica si el segmento  $S$  es vertical. Por ejemplo,

```
vertical(seg(punto(1,2), punto(1,3))) => Sí
vertical(seg(punto(1,2), punto(4,2))) => No
horizontal(seg(punto(1,2), punto(1,3))) => No
horizontal(seg(punto(1,2), punto(4,2))) => Sí
```

- Programa: `ver_hor.pl`

```
horizontal(seg(punto(X,Y), punto(X1,Y))).
vertical(seg(punto(X,Y), punto(X,Y1))).
```

## 1.2. Consultas sobre segmentos

### Consultas sobre segmentos verticales y horizontales

- ¿Es vertical el segmento que une los puntos  $(1, 1)$  y  $(1, 2)$ ?
 

```
?- vertical(seg(punto(1,1), punto(1,2))).
Yes
```
- ¿Es vertical el segmento que une los puntos  $(1, 1)$  y  $(2, 2)$ ?
 

```
?- vertical(seg(punto(1,1), punto(2,2))).
No
```
- ¿Hay algún  $Y$  tal que el segmento que une los puntos  $(1, 1)$  y  $(2, Y)$  sea vertical?
 

```
?- vertical(seg(punto(1,1), punto(2,Y))).
No
```
- ¿Hay algún  $X$  tal que el segmento que une los puntos  $(1, 2)$  y  $(X, 3)$  sea vertical?

```
?- vertical(seg(punto(1,2),punto(X,3))).
X = 1 ;
No
```

- ¿Hay algún Y tal que el segmento que une los puntos (1,1) y (2,Y) sea horizontal?

```
?- horizontal(seg(punto(1,1),punto(2,Y))).
Y = 1 ;
No
```

- ¿Para qué puntos el segmento que comienza en (2,3) es vertical?

```
?- vertical(seg(punto(2,3),P)).
P = punto(2, _G459) ;
No
```

- ¿Hay algún segmento que sea horizontal y vertical?

```
?- vertical(S),horizontal(S).
S = seg(punto(_G444, _G445),
        punto(_G444, _G445)) ;
No
?- vertical(_),horizontal(_).
Yes
```

## 2. Base de datos familiar

### 2.1. Representación de la base de datos familiar

#### Descripción de la familia 1

- el padre es Tomás García Pérez, nacido el 7 de Mayo de 1950, trabaja de profesor y gana 75 euros diarios
- la madre es Ana López Ruiz, nacida el 10 de marzo de 1952, trabaja de médica y gana 100 euros diarios
- el hijo es Juan García López, nacido el 5 de Enero de 1970, estudiante
- la hija es María García López, nacida el 12 de Abril de 1972, estudiante

**Representación de la familia 1**

```
familia(persona([tomas,garcia,perez],
               fecha(7,mayo,1950),
               trabajo(profesor,75)),
        persona([ana,lopez,ruiz],
               fecha(10,marzo,1952),
               trabajo(medica,100)),
        [ persona([juan,garcia,lopez],
                 fecha(5,enero,1970),
                 estudiante),
          persona([maria,garcia,lopez],
                 fecha(12,abril,1972),
                 estudiante) ]).
```

**Descripción de la familia 2**

- el padre es José Pérez Ruiz, nacido el 6 de Marzo de 1953, trabaja de pintor y gana 150 euros/día
- la madre es Luisa Gálvez Pérez, nacida el 12 de Mayo de 1954, es médica y gana 100 euros/día
- un hijo es Juan Luis Pérez Pérez, nacido el 5 de Febrero de 1980, estudiante
- una hija es María José Pérez Pérez, nacida el 12 de Junio de 1982, estudiante
- otro hijo es José María Pérez Pérez, nacido el 12 de Julio de 1984, estudiante

**Representación de la familia 2**

```
familia(persona([jose,perez,ruiz],
               fecha(6,marzo,1953),
               trabajo(pintor,150)),
        persona([luisa,galvez,perez],
               fecha(12,mayo,1954),
               trabajo(medica,100)),
        [ persona([juan_luis,perez,perez],
                 fecha(5,febrero,1980), estudiante),
          persona([maria_jose,perez,perez],
                 fecha(12,junio,1982), estudiante),
          persona([jose_maria,perez,perez],
                 fecha(12,julio,1984), estudiante)]).
```

## 2.2. Consultas a la base de datos familiar

### Consultas a la base de datos familiar

- ¿Existe alguna familia sin hijos?

```
?- familia(_,_, []).
No
```

- ¿Existe alguna familia con tres hijos?

```
?- familia(_,_,[_,_,_]).
Yes
```

- ¿Existe alguna familia con cuatro hijos?

```
?- familia(_,_,[_,_,_,_]).
No
```

- Buscar los nombres de los padres de familia con tres hijos

```
?- familia(persona(NP,_,_),_,[_,_,_]).
NP = [jose, perez, ruiz] ;
No
```

## 2.3. Extensiones y consultas a la base de datos familiar

### Hombres casados

- `casado(X)` se verifica si X es un hombre casado.

```
casado(X) :- familia(X,_,_).
```

- Consulta:

```
?- casado(X).
X = persona([tomas, garcia, perez],
            fecha(7, mayo, 1950),
            trabajo(profesor, 75)) ;
X = persona([jose, perez, ruiz],
            fecha(6, marzo, 1953),
            trabajo(pintor, 150)) ;
No
```



## Mujeres casadas

- `casada(X)` se verifica si X es una mujer casada.

```
casada(X) :- familia(_,X,_).
```

- Consulta:

```
?- casada(X).
X = persona([ana, lopez, ruiz],
            fecha(10, marzo, 1952),
            trabajo(medica, 100)) ;
X = persona([luisa, galvez, perez],
            fecha(12, mayo, 1954),
            trabajo(medica, 100)) ;
No
```

## Recuperación de la información

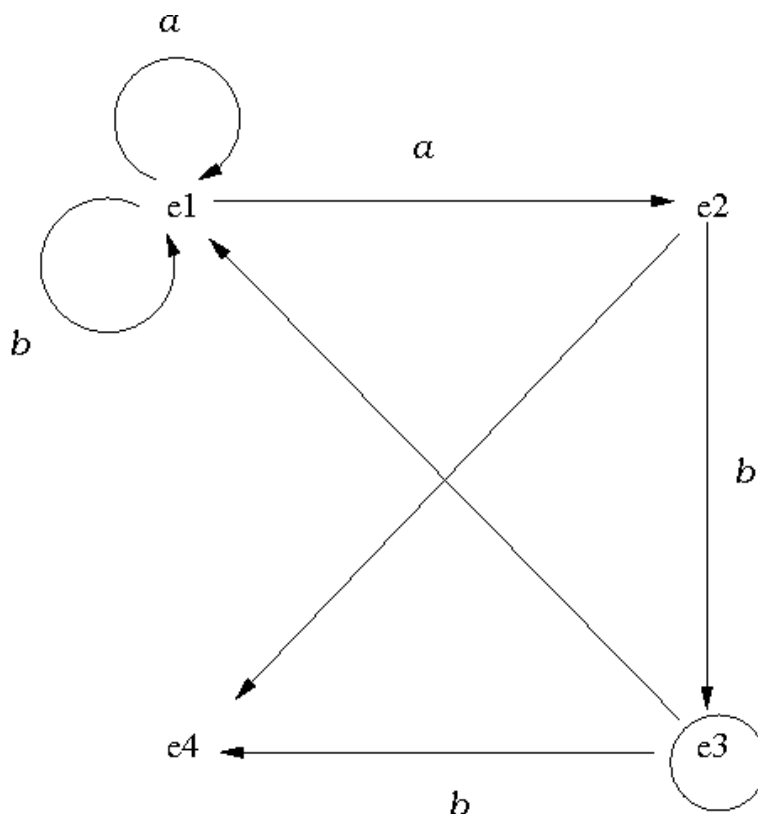
- Buscar los nombres de las mujeres casadas que trabajan.

```
?- casada(persona([N,_,_],_,trabajo(_,_))).
N = ana ;
N = luisa ;
No
```

## 3. Autómatas no deterministas

### 3.1. Representación de un autómata no determinista

Ejemplo de autómata no determinista (con estado final e3)



### Representación de un autómata (automata.pl)

- `final(E)` se verifica si E es el estado final.

```
final(e3).
```

- `trans(E1,X,E2)` se verifica si se puede pasar del estado E1 al estado E2 usando la letra X.

```
trans(e1,a,e1).   trans(e1,a,e2).   trans(e1,b,e1).
trans(e2,b,e3).
trans(e3,b,e4).
```

- `nulo(E1,E2)` se verifica si se puede pasar del estado E1 al estado E2 mediante un movimiento nulo.

```
nulo(e2,e4).
nulo(e3,e1).
```

### 3.2. Simulación de los autómatas no deterministas

#### Simulación de los autómatas no deterministas

- `acepta(E,L)` se verifica si el autómata, a partir del estado `E` acepta la lista `L`. Por ejemplo,

```

acepta(e1, [a, a, a, b]) => Sí
acepta(e2, [a, a, a, b]) => No

```

```

acepta(E, []) :-
    final(E).
acepta(E, [X|L]) :-
    trans(E, X, E1),
    acepta(E1, L).
acepta(E, L) :-
    nulo(E, E1),
    acepta(E1, L).

```

### 3.3. Consultas al autómata

#### Consultas al autómata

- Determinar si el autómata acepta la lista `[a, a, a, b]`

```

?- acepta(e1, [a, a, a, b]).
Yes

```

- Determinar los estados a partir de los cuales el autómata acepta la lista `[a, b]`

```

?- acepta(E, [a, b]).
E=e1 ;
E=e3 ;
No

```

- Determinar las palabras de longitud 3 aceptadas por el autómata a partir del estado `e1`

```

?- acepta(e1, [X, Y, Z]).
X = a   Y = a   Z = b ;
X = b   Y = a   Z = b ;
No

```

## 4. Problema de planificación

### 4.1. Representación del problema del mono

#### Representación del problema del mono

- Especificación: Un mono se encuentra en la puerta de una habitación. En el centro de la habitación hay un plátano colgado del techo. El mono está hambriento y desea coger el plátano, pero no lo alcanza desde el suelo. En la ventana de la habitación hay una silla que el mono puede usar. El mono puede realizar las siguientes acciones: pasear de un lugar a otro de la habitación, empujar la silla de un lugar a otro de la habitación (si está en el mismo lugar que la silla), subirse en la silla (si está en el mismo lugar que la silla) y coger el plátano (si está encima de la silla en el centro de la habitación).
- Representación: estado (PM, AM, PS, MM)
  - PM posición del mono (puerta, centro o ventana)
  - AM apoyo del mono (suelo o silla)
  - PS posición de la silla (puerta, centro o ventana)
  - MM mano del mono (con o sin) plátano

#### Acciones en el problema del mono

- `movimiento(E1, A, E2)` se verifica si E2 es el estado que resulta de realizar la acción A en el estado E1.

```

movimiento(estado(centro,silla,centro,sin),
           coger,
           estado(centro,silla,centro,con)).
movimiento(estado(X,suelo,X,U),
           subir,
           estado(X,silla,X,U)).
movimiento(estado(X1,suelo,X1,U),
           empujar(X1,X2),
           estado(X2,suelo,X2,U)).
movimiento(estado(X,suelo,Z,U),
           pasear(X,Z),
           estado(Z,suelo,Z,U)).

```

### 4.2. Solución del problema del mono

#### Solución del problema del mono

- `solución(E,S)` se verifica si `S` es una sucesión de acciones que aplicadas al estado `E` permiten al mono coger el plátano. Por ejemplo,

```
?- solución(estado(puerta,suelo,ventana,sin),L).
L = [pasear(puerta, ventana),
     empujar(ventana, centro),
     subir,
     coger]
```

```
solución(estado(_,_,_,con), []).
solución(E1, [A|L]) :-
    movimiento(E1,A,E2),
    solución(E2,L).
```

El grafo de deducción se muestra en la figura 1

## Bibliografía

### Bibliografía

1. I. Bratko *Prolog Programming for Artificial Intelligence (3 ed.)* (Addison–Wesley, 2001)
  - Cap. 2: “Syntax and meaning of Prolog programs”
  - Cap. 4: “Using Structures: Example Programs”
2. T. Van Le *Techniques of Prolog Programming* (John Wiley, 1993)
  - Cap. 2: “Declarative Prolog programming”.
3. W.F. Clocksin y C.S. Mellish *Programming in Prolog (Fourth Edition)* (Springer Verlag, 1994)
  - Cap. 3: “Using Data Structures”

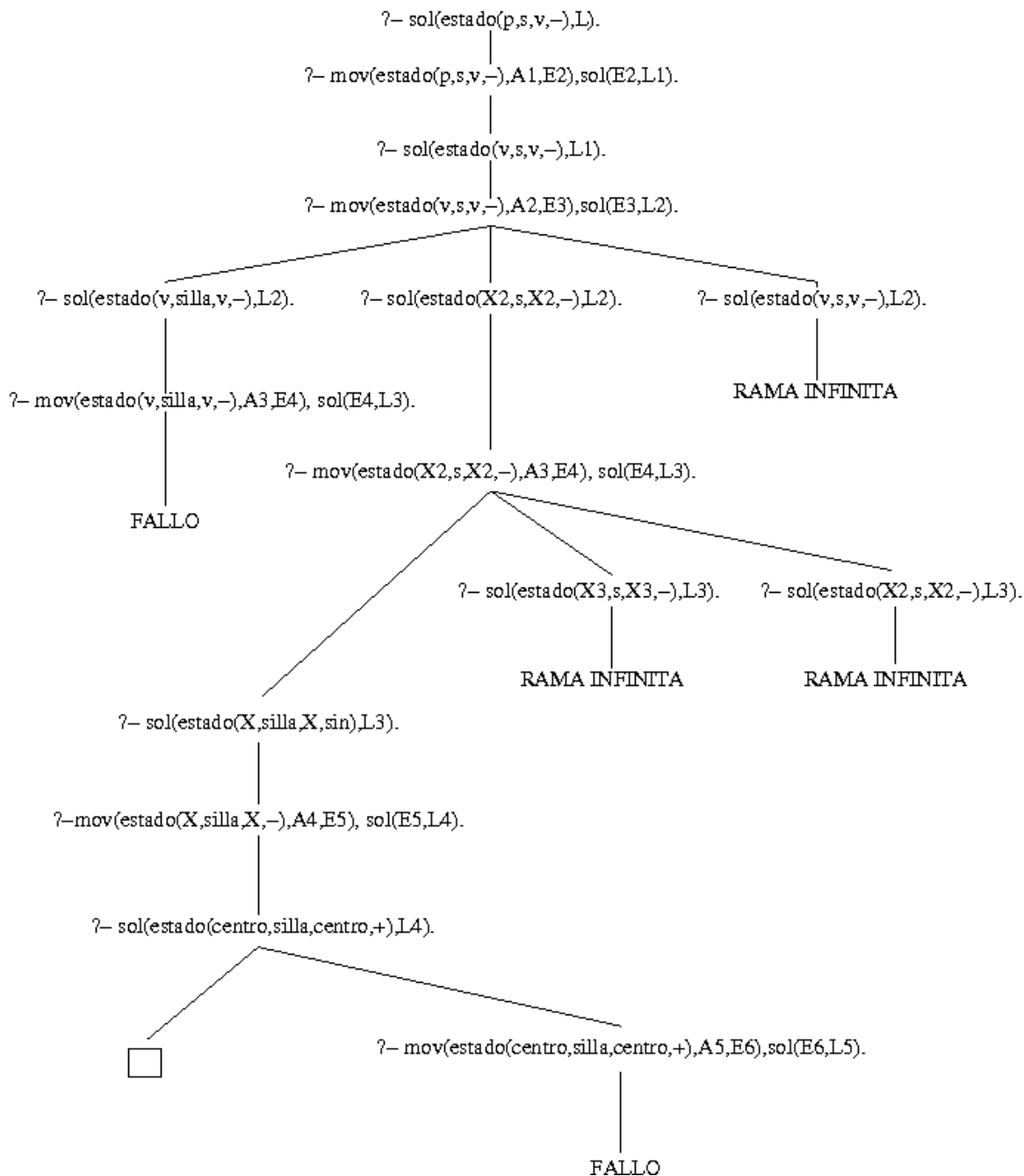


Figura 1: Grafo de deducción del problema del mono

## **Capítulo 9**

### **Retroceso, corte y negación**

# Programación declarativa (2008–09)

Tema 9: Retroceso, corte y negación

José A. Alonso Jiménez

## Índice

<b>1. Control mediante corte</b>	<b>1</b>
1.1. Control mediante corte . . . . .	1
1.2. Ejemplos usando el corte . . . . .	2
<b>2. Negación como fallo</b>	<b>3</b>
2.1. Definición de la negación como fallo . . . . .	3
2.2. Programas con negación como fallo . . . . .	3
<b>3. El condicional</b>	<b>7</b>



# 1. Control mediante corte

## 1.1. Control mediante corte

### Ejemplo de nota sin corte

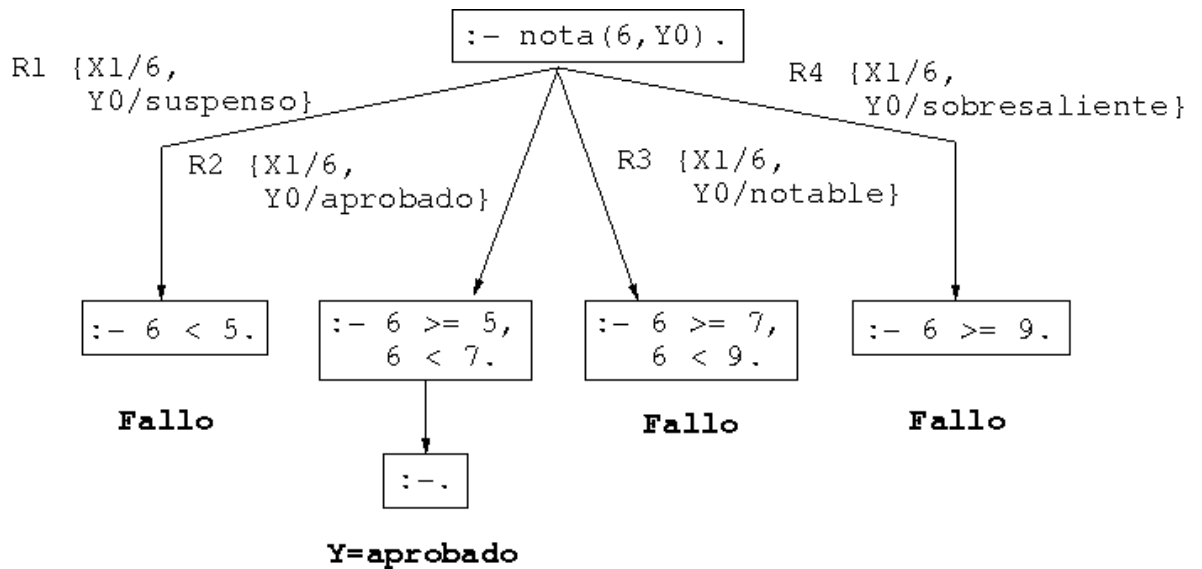
- nota(X, Y)** se verifica si Y es la calificación correspondiente a la nota X; es decir, Y es suspenso si X es menor que 5, Y es aprobado si X es mayor o igual que 5 pero menor que 7, Y es notable si X es mayor que 7 pero menor que 9 e Y es sobresaliente si X es mayor que 9. Por ejemplo,

```
?- nota(6,Y).
Y = aprobado;
No
```

```
nota(X,suspenso)      :- X < 5.
nota(X,aprobado)     :- X >= 5, X < 7.
nota(X,notable)      :- X >= 7, X < 9.
nota(X,sobresaliente) :- X >= 9.
```

### Deducción en el ejemplo sin corte

- Árbol de deducción de `?- nota(6,Y).`



### Ejemplo de nota con cortes

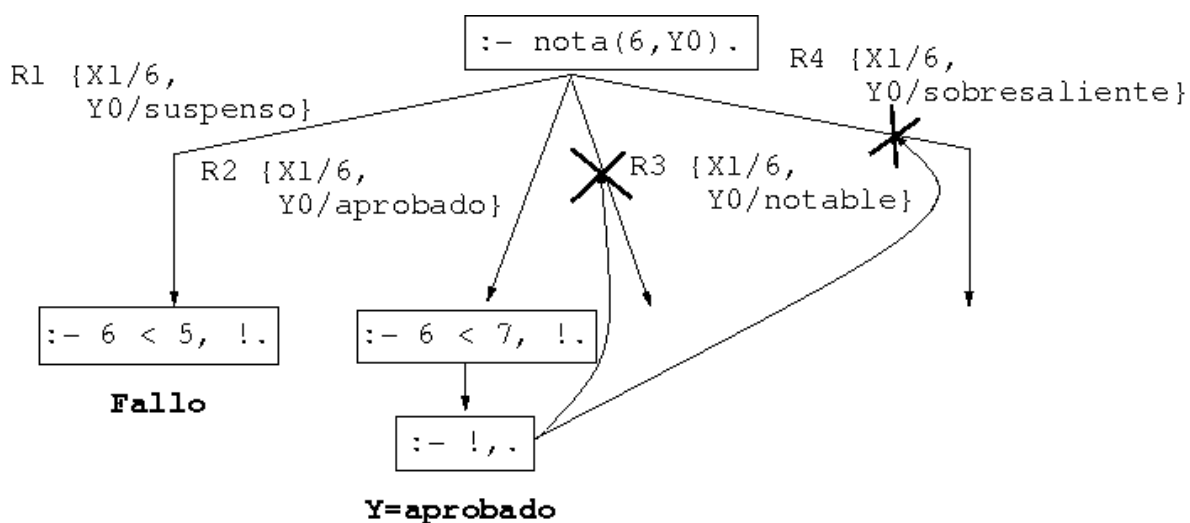
- Definición de nota con cortes

```

nota(X,suspenso)      :- X < 5, !.
nota(X,aprobado)     :- X < 7, !.
nota(X,notable)      :- X < 9, !.
nota(X,sobresaliente).

```

### Deducción en el ejemplo con cortes



- ¿Un 6 es un sobresaliente?

```

?- nota(6,sobresaliente).
Yes

```

## 1.2. Ejemplos usando el corte

### Uso de corte para respuesta única

- Diferencia entre `member` y `memberchk`

```

?- member(X, [a,b,a,c]), X=a.
X = a ;
X = a ;
No
?- memberchk(X, [a,b,a,c]), X=a.
X = a ;
No

```

- Definición de `member` y `memberchk`:

```
member(X, [X|_]).
member(X, [_|L]) :- member(X,L).

memberchk(X, [X|_]) :- !.
memberchk(X, [_|L]) :- memberchk(X,L).
```

## 2. Negación como fallo

### 2.1. Definición de la negación como fallo

#### Definición de la negación como fallo

- Definición de la negación como fallo `not`:

```
no(P) :- P, !, fail.           % No 1
no(P).                         % No 2
```

### 2.2. Programas con negación como fallo

#### Programa con negación

- Programa:

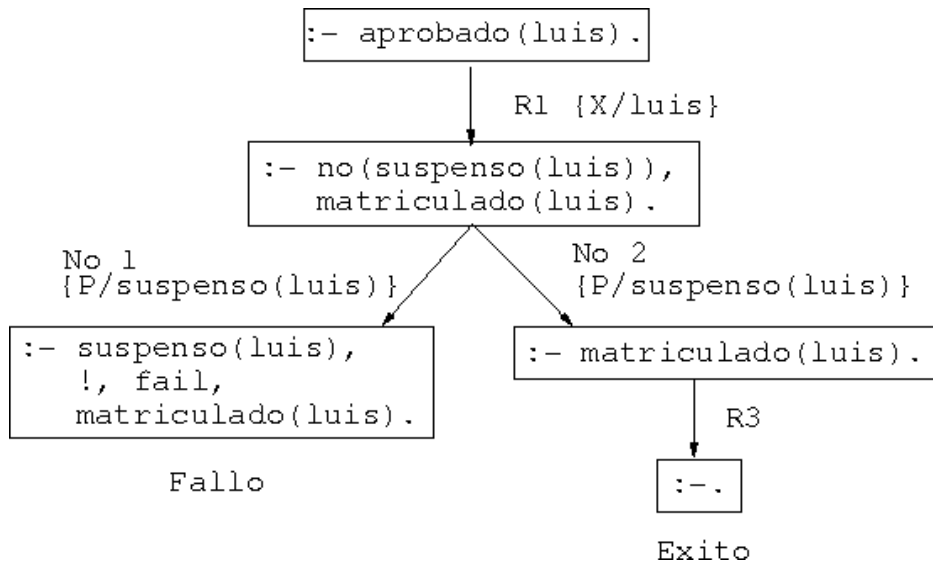
```
aprobado(X) :-                    % R1
    no(suspenso(X)),
    matriculado(X).
matriculado(juan).                % R2
matriculado(luis).                % R3
suspenso(juan).                   % R4
```

- Consultas:

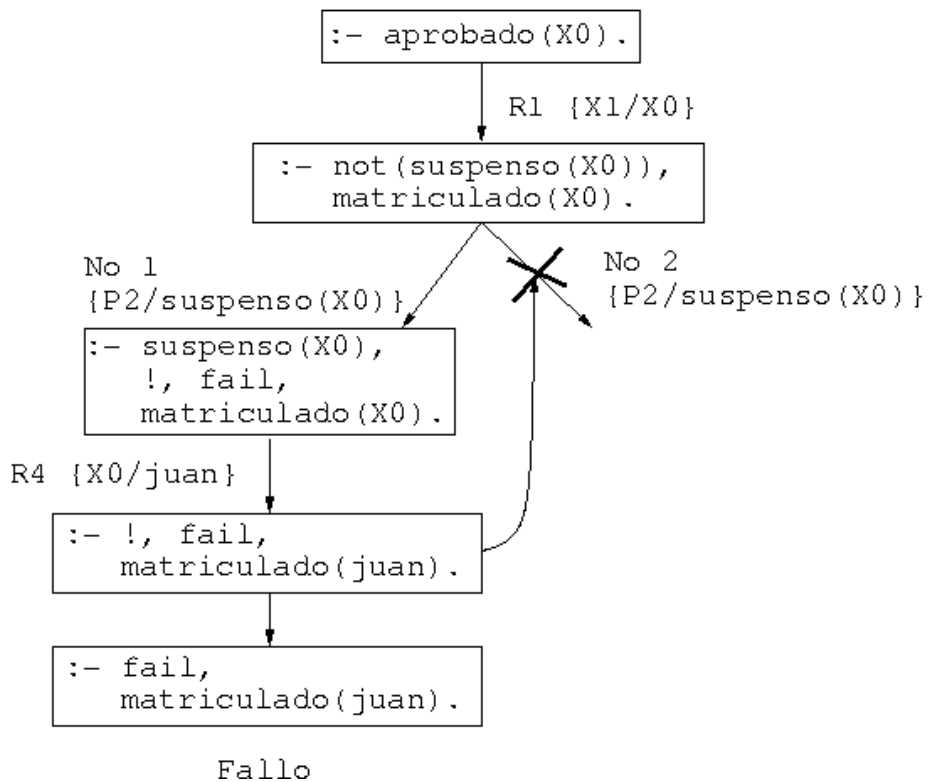
```
?- aprobado(luis).
Yes
```

```
?- aprobado(X).
No
```

Árbol de deducción de `?- aprobado(luis).`



Árbol de deducción de `?- aprobado(X).`



**Modificación del orden de los literales**

■ Programa:

```

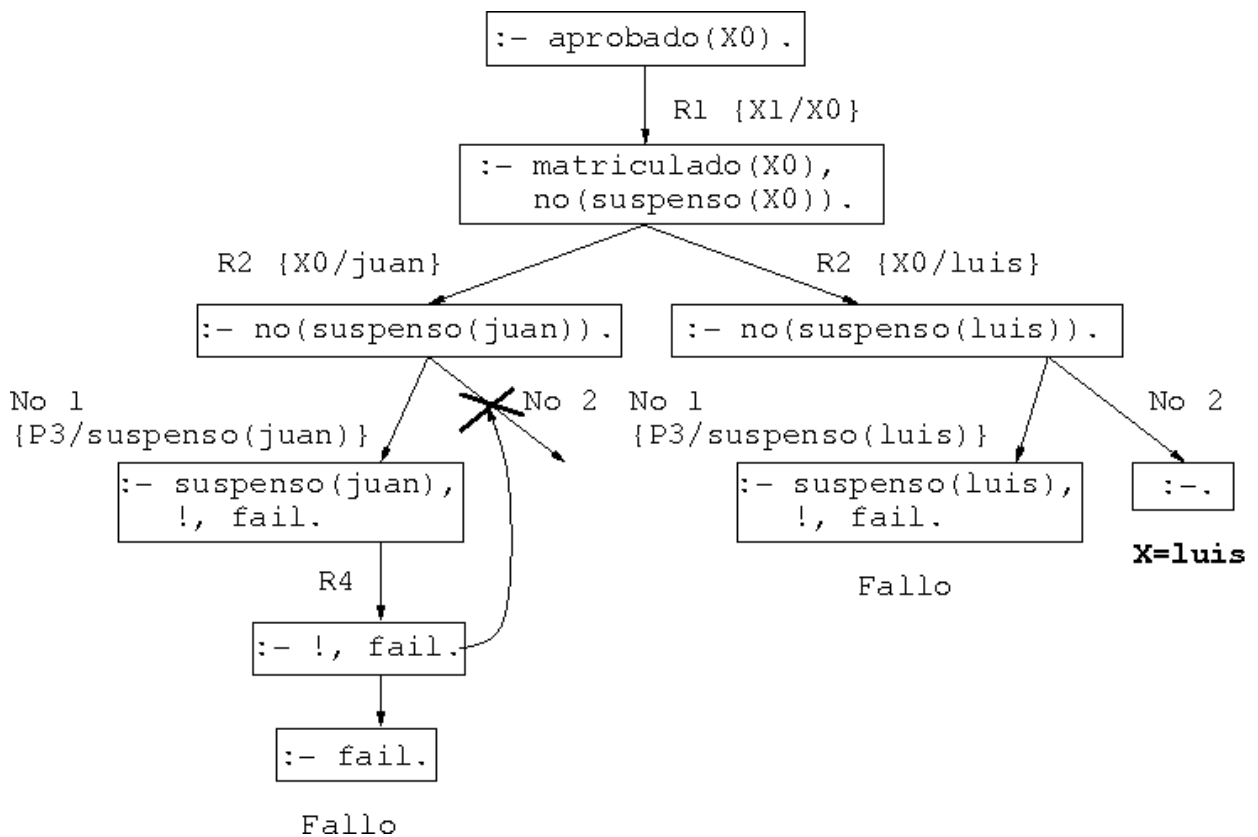
aprobado(X) :-          % R1
    matriculado(X),
    no(suspenso(X)).
matriculado(juan).    % R2
matriculado(luis).    % R3
suspenso(juan).       % R4
    
```

■ Consulta:

```

?- aprobado(X).
X = luis
Yes
    
```

Árbol de deducción de ?- aprobado(X).



Ejemplo de definición con not y con corte

- `borra(L1,X,L2)` se verifica si L2 es la lista obtenida eliminando los elementos de L1 unificables simultáneamente con X. Por ejemplo,

```
?- borra([a,b,a,c],a,L).
L = [b, c] ;
No
?- borra([a,Y,a,c],a,L).
Y = a
L = [c] ;
No
?- borra([a,Y,a,c],X,L).
Y = a
X = a
L = [c] ;
No
```

- Definición con not:

```
borra_1([],_,[]).
borra_1([X|L1],Y,L2) :-
    X=Y,
    borra_1(L1,Y,L2).
borra_1([X|L1],Y,[X|L2]) :-
    not(X=Y),
    borra_1(L1,Y,L2).
```

- Definición con corte:

```
borra_2([],_,[]).
borra_2([X|L1],Y,L2) :-
    X=Y, !,
    borra_2(L1,Y,L2).
borra_2([X|L1],Y,[X|L2]) :-
    % not(X=Y),
    borra_2(L1,Y,L2).
```

- Definición con corte y simplificada

```
borra_3([],_,[]).
borra_3([X|L1],X,L2) :-
    !,
    borra_3(L1,Y,L2).
borra_3([X|L1],Y,[X|L2]) :-
    % not(X=Y),
    borra_3(L1,Y,L2).
```

### 3. El condicional

#### Definición de nota con el condicional

- Definición de nota con el condicional:

```

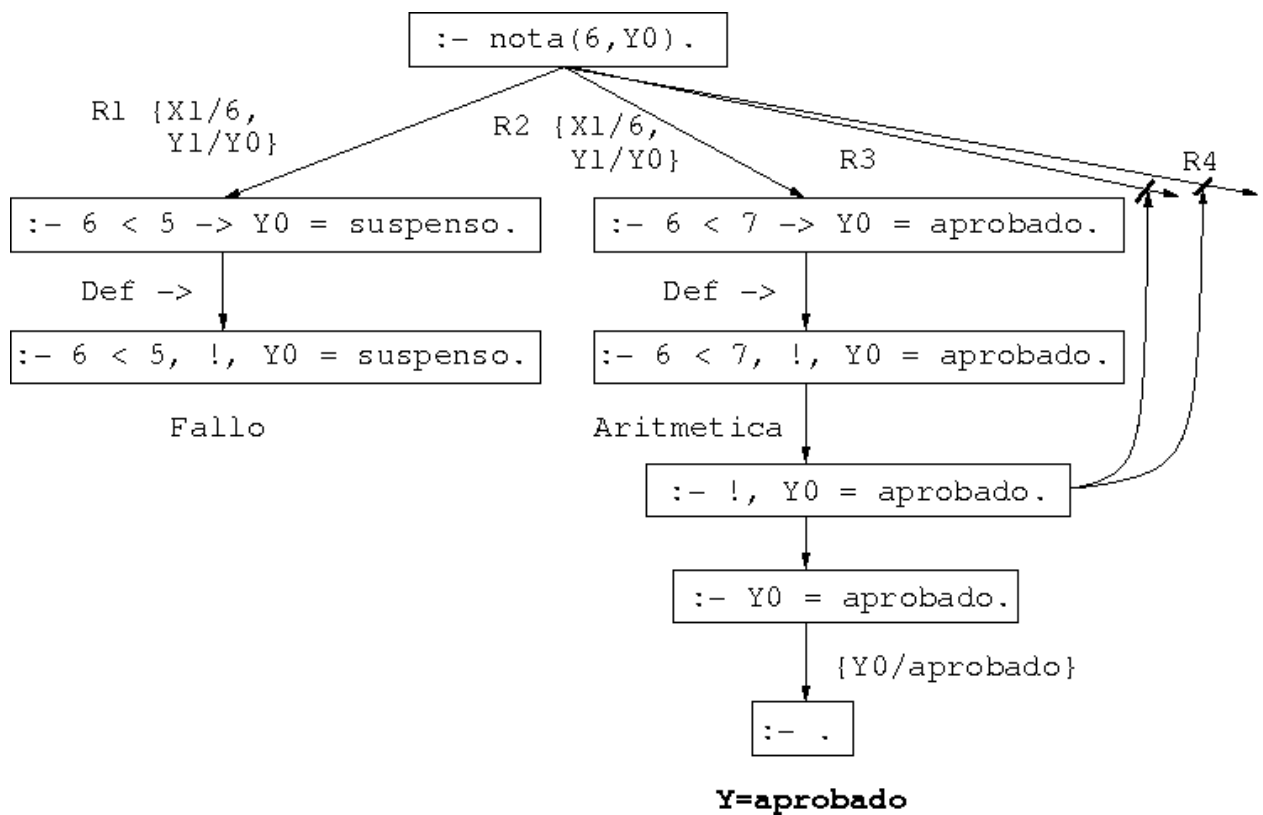
nota(X,Y) :-
  X < 5 -> Y = suspenso ;      % R1
  X < 7 -> Y = aprobado ;      % R2
  X < 9 -> Y = notable ;       % R3
  true -> Y = sobresaliente.    % R4
    
```

- Definición del condicional y verdad:

```

P -> Q :- P, !, Q.             % Def. ->
P -> Q.
true.                           % Def. true
    
```

#### Árbol de deducción de ?- nota(6,Y).



## Bibliografía

### Bibliografía

1. J.A. Alonso *Introducción a la programación lógica con Prolog*.
  - Cap. 7: “Control mediante corte”
  - Cap. 8: “Negación”
2. I. Bratko *Prolog Programming for Artificial Intelligence (3 ed.)* (Addison–Wesley, 2001)
  - Cap. 5: “Controlling backtracking”
3. W.F. Clocksin y C.S. Mellish *Programming in Prolog (Fourth Edition)* (Springer Verlag, 1994)
  - Cap. 4: “Backtracking and the cut”
4. L. Sterling y E. Shapiro *The Art of Prolog (2nd Edition)* (The MIT Press, 1994)
  - Cap. 11: “Cuts and negation”



# Capítulo 10

## Programación lógica de segundo orden

# Programación declarativa (2008–09)

Tema 10: Programación lógica de segundo orden

José A. Alonso Jiménez

## Índice

<b>1. Modificación de la base de conocimiento</b>	<b>1</b>
1.1. Predicados de modificación de la base de conocimiento . . . . .	1
1.2. Programa con modificación de la base de conocimiento . . . . .	2
<b>2. Todas las soluciones</b>	<b>3</b>
2.1. Predicados de todas las soluciones . . . . .	3
2.2. Operaciones con conjuntos . . . . .	5
<b>3. Transformación entre términos, átomos y listas</b>	<b>6</b>
3.1. Transformación entre términos y listas . . . . .	6
3.2. Transformaciones entre átomos y listas . . . . .	8
<b>4. Procedimientos aplicativos</b>	<b>8</b>
4.1. La relación <code>apply</code> . . . . .	8
4.2. La relación <code>maplist</code> . . . . .	9
<b>5. Relaciones sobre términos</b>	<b>9</b>
5.1. Predicados sobre tipos de término . . . . .	9
5.2. Comparación y ordenación de términos . . . . .	10

# 1. Modificación de la base de conocimiento

## 1.1. Predicados de modificación de la base de conocimiento

### Predicados `assert` y `retract`

- `assert(+Term)` inserta un hecho o una cláusula en la base de conocimientos. `Term` es insertado como última cláusula del predicado correspondiente.
- `retract(+Term)` elimina la primera cláusula de la base de conocimientos que unifica con `Term`.

```
|?- hace_frio.
|
|No
|?- assert(hace_frio).
|Yes
|?- hace_frio.
|
|Yes
|?- retract(hace_frio).
|Yes
|?- hace_frio.
|
|No
```

### El predicado `listing`

- `listing(+Pred)` lista las cláusulas en cuya cabeza aparece el predicado `Pred`, Por ejemplo,

```
|?- assert((gana(X,Y) :- rápido(X), lento(Y))).
|?- listing(gana).
|
|gana(A, B) :- rápido(A), lento(B).
|?- assert(rápido(juan)), assert(lento(jose)),
|   assert(lento(luis)).
|?- gana(X,Y).
|
|X = juan  Y = jose ; X = juan  Y = luis ;
|No
|?- retract(lento(X)).
|X = jose ; X = luis ;
|No
|?- gana(X,Y).
|
|No
```

### Los predicados `asserta` y `assertz`

- `asserta(+Term)` equivale a `assert/1`, pero `Term` es insertado como primera cláusula del predicado correspondiente.
- `assertz(+Term)` equivale a `assert/1`.

```
?- assert(p(a)),
   assertz(p(b)),
   asserta(p(c)).
Yes
?- p(X).

X = c ; X = a ; X = b ; No
?- listing(p).

p(c).
p(a).
p(b).
Yes
```

### Los predicados `retractall` y `abolish`

- `retractall(+C)` elimina de la base de conocimientos todas las cláusulas cuya cabeza unifica con `C`.
- `abolish(+SimbPred/+Aridad)` elimina de la base de conocimientos todas las cláusulas que en su cabeza aparece el símbolo de predicado `SimbPred/Aridad`.

```
?- assert(p(a)), assert(p(b)).
?- retractall(p(_)).
?- p(a).
No
?- assert(p(a)), assert(p(b)).
?- abolish(p/1).
?- p(a).
% [WARNING: Undefined predicate: 'p/1']
No
```

## 1.2. Programa con modificación de la base de conocimiento

### Programa con modificación de la base de conocimiento

- `crea_tabla` añade los hechos `producto(X,Y,Z)` donde `X` e `Y` son números de 0 a 9 y `Z` es el producto de `X` e `Y`. Por ejemplo,

```
?- crea_tabla.
Yes
?- listing(producto).
producto(0,0,0).
producto(0,1,0).
...
producto(9,8,72).
producto(9,9,81).
Yes
```

- Definición:

```
crea_tabla :-
    L = [0,1,2,3,4,5,6,7,8,9],
    member(X,L),
    member(Y,L),
    Z is X*Y,
    assert(producto(X,Y,Z)),
    fail.
crea_tabla.
```

- Calcular las descomposiciones de 6 en producto de dos números.

```
?- producto(A,B,6).
A=1 B=6 ;
A=2 B=3 ;
A=3 B=2 ;
A=6 B=1 ;
No
```

## 2. Todas las soluciones

### 2.1. Predicados de todas las soluciones

#### Lista de soluciones (findall)

- `findall(T,0,L)` se verifica si L es la lista de las instancias del término T que verifican el objetivo 0.

```
?- assert(clase(a,voc), assert(clase(b,con)),
    assert(clase(e,voc), assert(clase(c,con))).
?- findall(X,clase(X,voc),L).
X = _G331    L = [a, e]
```

```
?- findall(_X,clase(_X,_Clase),L).
L = [a, b, e, c]
?- findall(X,clase(X,vocal),L).
X = _G355    L = []
?- findall(X,(member(X,[c,b,c]),member(X,[c,b,a])),L).
X = _G373    L = [c, b, c]
?- findall(X,(member(X,[c,b,c]),member(X,[1,2,3])),L).
X = _G373    L = []
```

### Conjunto de soluciones (setof)

- `setof(T,0,L)` se verifica si L es la lista ordenada sin repeticiones de las instancias del término T que verifican el objetivo 0.

```
?- setof(X,clase(X,Clase),L).
X = _G343    Clase = voc    L = [a, e] ;
X = _G343    Clase = con    L = [b, c] ;
No
?- setof(X,Y^clase(X,Y),L).
X = _G379    Y = _G380    L = [a, b, c, e]
?- setof(X,clase(X,vocal),L).
No
?- setof(X,(member(X,[c,b,c]),member(X,[c,b,a])),L).
X = _G361    L = [b, c]
?- setof(X,(member(X,[c,b,c]),member(X,[1,2,3])),L).
No
```

### Multiconjunto de soluciones (bagof)

- `bagof(T,0,L)` se verifica si L es el multiconjunto de las instancias del término T que verifican el objetivo 0.

```
?- bagof(X,clase(X,Clase),L).
X = _G343    Clase = voc    L = [a, e] ;
X = _G343    Clase = con    L = [b, c] ;
No
?- bagof(X,Y^clase(X,Y),L).
X = _G379    Y = _G380    L = [a, b, e, c]
?- bagof(X,clase(X,vocal),L).
No
?- bagof(X,(member(X,[c,b,c]),member(X,[c,b,a])),L).
X = _G361    L = [c, b, c]
?- bagof(X,(member(X,[c,b,c]),member(X,[1,2,3])),L).
No
```

## 2.2. Operaciones con conjuntos

### Conjunto de todas las soluciones

- `setof0(T,0,L)` es como `setof` salvo en el caso en que ninguna instancia de `T` verifique `0`, en cuyo caso `L` es la lista vacía. Por ejemplo,

```
?- setof0(X,
      (member(X,[c,a,b]),member(X,[c,b,d])),
      L).
L = [b, c]
?- setof0(X,
      (member(X,[c,a,b]),member(X,[e,f])),
      L).
L = []
```

- Definición:

```
setof0(X,0,L) :- setof(X,0,L), !.
setof0(_,_,[]).
```

### Intersección y unión

- `intersección(S,T,U)` se verifica si `U` es la intersección de `S` y `T`. Por ejemplo,

```
?- intersección([1,4,2],[2,3,4],U).
U = [2,4]
```

```
intersección(S,T,U) :-
  setof0(X, (member(X,S), member(X,T)), U).
```

- `unión(S,T,U)` se verifica si `U` es la unión de `S` y `T`. Por ejemplo,

```
?- unión([1,2,4],[2,3,4],U).
U = [1,2,3,4]
```

```
unión(S,T,U) :-
  setof(X, (member(X,S); member(X,T)), U).
```

## Diferencia y conjunto potencia

- `diferencia(S,T,U)` se verifica si U es la diferencia de los conjuntos de S y T. Por ejemplo,

```
?- diferencia([5,1,2],[2,3,4],U).
U = [1,5]
```

```
diferencia(S,T,U) :-
    setof(X,(member(X,S),not(member(X,T))),U).
```

- `partes(X,L)` se verifica si L es el conjunto de las partes de X. Por ejemplo,

```
?- partes([a,b,c],L).
L = [[],[a],[a,b],[a,b,c],[a,c],[b],[b,c],[c]]
```

```
partes(X,L) :-
    setof(Y,subconjunto(Y,X),L).
```

## Cálculo de subconjuntos

- `subconjunto(-L1,+L2)` se verifica si L1 es un subconjunto de L2. Por ejemplo,

```
?- subconjunto(L,[a,b]).
L = [a, b] ;
L = [a] ;
L = [b] ;
L = [] ;
No
```

```
subconjunto([],[]).
subconjunto([X|L1],[X|L2]) :-
    subconjunto(L1,L2).
subconjunto(L1,[_|L2]) :-
    subconjunto(L1,L2).
```

## 3. Transformación entre términos, átomos y listas

### 3.1. Transformación entre términos y listas

#### Relación de transformación entre términos y listas



- `?T =.. ?L` se verifica si L es una lista cuyo primer elemento es el functor del término T y los restantes elementos de L son los argumentos de T. Por ejemplo,

```
?- padre(juan,luis) =.. L.
L = [padre, juan, luis]
?- T =.. [padre, juan, luis].
T = padre(juan,luis)
```

### Programa con transformación entre términos y listas

- `alarga(+F1,+N,-F2)` se verifica si F1 y F2 son figuras del mismo tipo y el tamaño de F1 es el de F2 multiplicado por N,

```
?- alarga(triángulo(3,4,5),2,F).
F = triángulo(6, 8, 10)
?- alarga(cuadrado(3),2,F).
F = cuadrado(6)
```

```
alarga(Figura1,Factor,Figura2) :-
    Figura1 =.. [Tipo|Arg1],
    multiplica_lista(Arg1,Factor,Arg2),
    Figura2 =.. [Tipo|Arg2].

multiplica_lista([],_,[]).
multiplica_lista([X1|L1],F,[X2|L2]) :-
    X2 is X1*F, multiplica_lista(L1,F,L2).
```

### Las relaciones functor y arg

- `functor(T,F,A)` se verifica si F es el functor del término T y A es su aridad.
- `arg(N,T,A)` se verifica si A es el argumento del término T que ocupa el lugar N.

```
?- functor(g(b,c,d),F,A).
F = g
A = 3
?- functor(T,g,2).
T = g(_G237,_G238)
?- arg(2,g(b,c,d),X).
X = c
?- functor(T,g,3),arg(1,T,b),arg(2,T,c).
T = g(b, c, _G405)
```

### 3.2. Transformaciones entre átomos y listas

**Relación de transformación entre átomos y listas:** `name`

- `name(A,L)` se verifica si L es la lista de códigos ASCII de los caracteres del átomo A. Por ejemplo,

```
?- name(bandera,L).
L = [98, 97, 110, 100, 101, 114, 97]
?- name(A,[98, 97, 110, 100, 101, 114, 97]).
A = bandera
```

**Programa con transformación entre átomos y listas**

- `concatena_átomos(A1,A2,A3)` se verifica si A3 es la concatenación de los átomos A1 y A2. Por ejemplo,

```
?- concatena_átomos(pi,ojo,X).
X = piojo
```

```
concatena_átomos(A1,A2,A3) :-
    name(A1,L1),
    name(A2,L2),
    append(L1,L2,L3),
    name(A3,L3).
```

## 4. Procedimientos aplicativos

### 4.1. La relación `apply`

La relación `apply`

- `apply(T,L)` se verifica si es demostrable T después de aumentar el número de sus argumentos con los elementos de L; por ejemplo,

```
plus(2,3,X).                => X=5
apply(plus,[2,3,X]).        => X=5
apply(plus(2),[3,X]).      => X=5
apply(plus(2,3),[X]).      => X=5
apply(append([1,2]),[X,[1,2,3]]). => X=[3]
```

```
n_apply(Término,Lista) :-
    Término =.. [Pred|Arg1],
    append(Arg1,Lista,Arg2),
    Átomo =.. [Pred|Arg2],
    Átomo.
```

## 4.2. La relación maplist

### La relación maplist

- `maplist(P,L1,L2)` se verifica si se cumple el predicado `P` sobre los sucesivos pares de elementos de las listas `L1` y `L2`; por ejemplo,

```
?- succ(2,X).           => 3
?- succ(X,3).          => 2
?- maplist(succ,[2,4],[3,5]). => Yes
?- maplist(succ,[0,4],[3,5]). => No
?- maplist(succ,[2,4],Y).   => Y = [3,5]
?- maplist(succ,X,[3,5]).  => X = [2,4]
```

```
n_maplist(_,[],[]).
n_maplist(R,[X1|L1],[X2|L2]) :-
    apply(R,[X1,X2]),
    n_maplist(R,L1,L2).
```

## 5. Relaciones sobre términos

### 5.1. Predicados sobre tipos de término

#### Predicados sobre tipos de término

- `var(T)` se verifica si `T` es una variable.
- `atom(T)` se verifica si `T` es un átomo.
- `number(T)` se verifica si `T` es un número.
- `compound(T)` se verifica si `T` es un término compuesto.
- `atomic(T)` se verifica si `T` es una variable, átomo, cadena o número.

```

?- var(X1).           => Yes
?- atom(átomo).      => Yes
?- number(123).      => Yes
?- number(-25.14).   => Yes
?- compound(f(X,a)). => Yes
?- compound([1,2]).  => Yes
?- atomic(átomo).    => Yes
?- atomic(123).      => Yes

```

### Programa con predicados sobre tipos de término

- `suma_segura(X,Y,Z)` se verifica si `X` e `Y` son enteros y `Z` es la suma de `X` e `Y`. Por ejemplo,

```

?- suma_segura(2,3,X).
X = 5
Yes
?- suma_segura(7,a,X).
No
?- X is 7 + a.
% [WARNING: Arithmetic: 'a' is not a function]

```

```

suma_segura(X,Y,Z) :-
    number(X),
    number(Y),
    Z is X+Y.

```

## 5.2. Comparación y ordenación de términos

### Relaciones de comparación de términos

- `T1 = T2` se verifica si `T1` y `T2` son unificables.
- `T1 == T2` se verifica si `T1` y `T2` son idénticos.
- `T1 \== T2` se verifica si `T1` y `T2` no son idénticos.

```

?- f(X) = f(Y).
X = _G164
Y = _G164
Yes
?- f(X) == f(Y).
No

```

```
?- f(X) == f(X).
X = _G170
Yes
```

### Programa con comparación de términos

- `cuenta(A,L,N)` se verifique si `N` es el número de ocurrencias del átomo `A` en la lista `L`. Por ejemplo,

```
?- cuenta(a,[a,b,a,a],N).
N = 3
?- cuenta(a,[a,b,X,Y],N).
N = 1
```

```
cuenta(_, [], 0).
cuenta(A, [B|L], N) :-
    A == B, !,
    cuenta(A, L, M),
    N is M+1.
cuenta(A, [B|L], N) :-
    % A \== B,
    cuenta(A, L, N).
```

### Relaciones de ordenación de términos

- `T1 @< T2` se verifica si el término `T1` es anterior que `T2` en el orden de términos de Prolog.

```
?- ab @< ac.           => Yes
?- 21 @< 123.         => Yes
?- 12 @< a.           => Yes
?- g @< f(b).         => Yes
?- f(b) @< f(a,b).    => Yes
?- [a,1] @< [a,3].    => Yes
```

- `sort(+L1, -L2)` se verifica si `L2` es la lista obtenida ordenando de manera creciente los distintos elementos de `L1` y eliminando las repeticiones.

```
?- sort([c4,2,a5,2,c3,a5,2,a5],L).
L = [2, a5, c3, c4]
```

## Bibliografía

### Bibliografía

1. J.A. Alonso *Introducción a la programación lógica con Prolog*.
  - Cap. 10 “Predicados sobre tipos de término”
  - Cap. 11 “Comparación y ordenación de términos”
  - Cap. 12 “Procesamiento de términos”
  - Cap. 13 “Procedimientos aplicativos”
  - Cap. 14 “Todas las soluciones”
2. I. Bratko *Prolog Programming for Artificial Intelligence (3 ed.)* (Addison–Wesley, 2001)
  - Cap. 7: “More Built–in Procedures”
3. T. Van Le *Techniques of Prolog Programming* (John Wiley, 1993)
  - Cap. 6: “Advanced programming techniques and data structures”
4. W.F. Clocksin y C.S. Mellish *Programming in Prolog (Fourth Edition)* (Springer Verlag, 1994)
  - Cap. 6: “Built–in Predicates”

## **Capítulo 11**

# **Estilo y eficiencia en programación lógica**

# Programación declarativa (2008–09)

Tema 11: Estilo y eficiencia en programación lógica

José A. Alonso Jiménez

## Índice

<b>1. Principios generales de buena programación</b>	<b>1</b>
1.1. Criterios para juzgar la bondad de un programa . . . . .	1
1.2. Heurísticas para la programación (Ref. Polya) . . . . .	1
<b>2. Métodos de mejora de la eficiencia</b>	<b>1</b>
2.1. Orden de los literales . . . . .	1
2.2. Generación y prueba . . . . .	6
2.3. Uso de listas o términos compuestos . . . . .	9
2.4. Uso de la unificación implícita . . . . .	9
2.5. Acumuladores . . . . .	10
2.6. Uso de lemas . . . . .	11
2.7. Determinismo . . . . .	13
2.8. Añadir al principio . . . . .	14
2.9. Listas de diferencias . . . . .	15



# 1. Principios generales de buena programación

## 1.1. Criterios para juzgar la bondad de un programa

### Criterios para juzgar la bondad de un programa

- Corrección.
- Eficiencia.
- Facilidad de lectura.
- Modificabilidad: Modular y transparente.
- Robustez.
- Documentación.

## 1.2. Heurísticas para la programación (Ref. Polya)

### Heurísticas para la programación (Ref. Polya)

1. Entender el problema.
2. Diseñar una solución.
3. Escribir el programa.
4. Revisar la solución

# 2. Métodos de mejora de la eficiencia

## 2.1. Orden de los literales

### Orden de los literales y corrección

- Programas:

```
siguiente(a,1).
siguiente(1,b).

sucesor_1(X,Y) :-
    siguiente(X,Y).
sucesor_1(X,Y) :-
    siguiente(X,Z),
    sucesor_1(Z,Y).
```

```
|?- sucesor_1(X,Y).
```

```
|X = a Y = 1 ; X = 1 Y = b ; X = a Y = b ; No
|?- findall(X-Y,sucesor_1(X,Y),L).
```

```
|L = [a-1, 1-b, a-b]
```

#### ■ Programas:

```
siguiente(a,1).
siguiente(1,b).

sucesor_2(X,Y) :-
    siguiente(X,Y).
sucesor_2(X,Y) :-
    sucesor_2(Z,Y),
    siguiente(X,Z).
```

```
|?- sucesor_2(X,Y).
```

```
|X = a Y = 1 ; X = 1 Y = b ; X = a Y = b ;
|ERROR: Out of local stack
```

### Orden de los literales y eficiencia

#### ■ Orden de los literales y eficiencia

```
?- listing(número).
número(1). número(2). ... número(1000).

?- listing(múltiplo_de_100).
múltiplo_de_100(100). ... múltiplo_de_100(1000).

?- time((número(_N),múltiplo_de_100(_N))).
101 inferences in 0.00 seconds
Yes

?- time((múltiplo_de_100(_N),número(_N))).
2 inferences in 0.00 seconds
Yes
```

## Combinaciones

- `combinación(+L1,+N,-L2)` se verifica si `L2` es una combinación `N`-aria de `L1`. Por ejemplo,

```
?- combinación([a,b,c],2,L).
L = [a, b] ;   L = [a, c] ;   L = [b, c] ;   No
```

```
combinación_1(L1,N,L2) :-
    subconjunto(L2,L1),
    length(L2,N).

combinación_2(L1,N,L2) :-
    length(L2,N),
    subconjunto(L2,L1).

combinación(L1,N,L2) :-
    combinación_2(L1,N,L2).
```

- `combinaciones(+L1,+N,-L2)` se verifica si `L2` es la lista de las combinaciones `N`-arias de `L1`. Por ejemplo,

```
?- combinaciones([a,b,c],2,L).
L = [[a, b], [a, c], [b, c]]
```

```
combinaciones_1(L1,N,L2) :-
    findall(L,combinación_1(L1,N,L),L2).

combinaciones_2(L1,N,L2) :-
    findall(L,combinación_2(L1,N,L),L2).

combinaciones(L1,N,L2) :-
    combinaciones_2(L1,N,L2).
```

## Comparación de eficiencia de combinaciones

```
?- findall(_N,between(1,6,_N),_L1),
   time(combinaciones_1(_L1,2,_L2)),
   time(combinaciones_2(_L1,2,_L2)).
429 inferences in 0.00 seconds
92 inferences in 0.00 seconds
?- findall(_N,between(1,12,_N),_L1),
```

```

    time(combinaciones_1(_L1,2,_L2)),
    time(combinaciones_2(_L1,2,_L2)).
28,551 inferences in 0.01 seconds
    457 inferences in 0.00 seconds
?- findall(_N,between(1,24,_N),_L1),
    time(combinaciones_1(_L1,2,_L2)),
    time(combinaciones_2(_L1,2,_L2)).
117,439,971 inferences in 57.59 seconds
    2,915 inferences in 0.00 seconds

```

## Permutaciones

- `select(?X,?L1,?L2)` se verifica si X es un elemento de la lista L1 y L2 es la lista de los restantes elementos. Por ejemplo,

```

?- select(X,[a,b,c],L).
X = a    L = [b, c] ;
X = b    L = [a, c] ;
X = c    L = [a, b] ;
No

?- select(a,L,[b,c]).
L = [a, b, c] ;
L = [b, a, c] ;
L = [b, c, a] ;
No

```

- `permutación(+L1,-L2)` se verifica si L2 es una permutación de L1. Por ejemplo,

```

?- permutación([a,b,c],L).
L = [a, b, c] ; L = [a, c, b] ;
L = [b, a, c] ; L = [b, c, a] ;
L = [c, a, b] ; L = [c, b, a] ;
No

```

```

permutación([],[]).
permutación(L1,[X|L2]) :-
    select(X,L1,L3),
    permutación(L3,L2).

```

Predefinida `permutation`.

## Variaciones

- `variación(+L1,+N,-L2)` se verifica si `L2` es una variación `N`-aria de `L1`. Por ejemplo,

```
?- variación([a,b,c],2,L).
L=[a,b];L=[a,c];L=[b,a];L=[b,c];L=[c,a];L=[c,b];No
```

```
variación_1(L1,N,L2) :-
    combinación(L1,N,L3), permutación(L3,L2).

variación_2(_,0,[]).
variación_2(L1,N,[X|L2]) :-
    N > 0, M is N-1,
    select(X,L1,L3),
    variación_2(L3,M,L2).

variación(L1,N,L2) :- variación_2(L1,N,L2).
```

- `variaciones(+L1,+N,-L2)` se verifica si `L2` es la lista de las variaciones `N`-arias de `L1`. Por ejemplo,

```
?- variaciones([a,b,c],2,L).
L = [[a,b],[a,c],[b,a],[b,c],[c,a],[c,b]]
```

```
variaciones_1(L1,N,L2) :-
    setof(L,variación_1(L1,N,L),L2).

variaciones_2(L1,N,L2) :-
    setof(L,variación_2(L1,N,L),L2).

variaciones(L1,N,L2) :-
    variaciones_2(L1,N,L2).
```

## Comparación de eficiencia de variaciones

```
?- findall(N,between(1,100,N),L1),
   time(variaciones_1(L1,2,L2)),
   time(variaciones_2(L1,2,L2)).
221,320 inferences in 0.27 seconds
40,119 inferences in 0.11 seconds
?- findall(N,between(1,200,N),L1),
```

```

    time(variaciones_1(L1,2,L2)),
    time(variaciones_2(L1,2,L2)).
1,552,620 inferences in 2.62 seconds
    160,219 inferences in 0.67 seconds
?- findall(N,between(1,400,N),L1),
    time(variaciones_1(L1,2,L2)),
    time(variaciones_2(L1,2,L2)).
11,545,220 inferences in 19.02 seconds
    640,419 inferences in 2.51 seconds

```

## 2.2. Generación y prueba

### Ordenación por generación y prueba

- ordenación(+L1, -L2) se verifica si L2 es la lista obtenida ordenando la lista L1 en orden creciente. Por ejemplo,

```

?- ordenación([2,1,a,2,b,3],L).
L = [a,b,1,2,2,3]

```

```

ordenación(L,L1) :-
    permutación(L,L1),
    ordenada(L1).

```

```

ordenada([]).
ordenada([_]).
ordenada([X,Y|L]) :-
    X @=< Y,
    ordenada([Y|L]).

```

### Ordenación por selección

```

ordenación_por_selección(L1,[X|L2]) :-
    selecciona_menor(X,L1,L3),
    ordenación_por_selección(L3,L2).
ordenación_por_selección([],[]).

selecciona_menor(X,L1,L2) :-
    select(X,L1,L2),
    not((member(Y,L2), Y @< X)).

```

### Ordenación por divide y vencerás

```
ordenación_rápida([], []).
ordenación_rápida([X|R], Ordenada) :-
    divide(X, R, Menores, Mayores),
    ordenación_rápida(Menores, Menores_ord),
    ordenación_rápida(Mayores, Mayores_ord),
    append(Menores_ord, [X|Mayores_ord], Ordenada).

divide(_, [], [], []).
divide(X, [Y|R], [Y|Menores], Mayores) :-
    Y @< X, !,
    divide(X, R, Menores, Mayores).
divide(X, [Y|R], Menores, [Y|Mayores]) :-
    \% Y @>= X,
    divide(X, R, Menores, Mayores).
```

### Ordenación: comparación de eficiencia

Comparación de la ordenación de la lista [N, N-1, N-2, . . . , 2, 1]

N	ordena	selección	rápida
1	5 inf 0.00 s	8 inf 0.00 s	5 inf 0.00 s
2	10 inf 0.00 s	19 inf 0.00 s	12 inf 0.00 s
4	80 inf 0.00 s	67 inf 0.00 s	35 inf 0.00 s
8	507,674 inf 0.33 s	323 inf 0.00 s	117 inf 0.00 s
16		1,923 inf 0.00 s	425 inf 0.00 s
32		13,059 inf 0.01 s	1,617 inf 0.00 s
64		95,747 inf 0.05 s	6,305 inf 0.00 s
128		732,163 inf 0.40 s	24,897 inf 0.01 s
256		5,724,163 inf 2.95 s	98,945 inf 0.05 s
512		45,264,899 inf 22.80 s	394,497 inf 0.49 s

### Cuadrado mágico por generación y prueba

- Enunciado: Colocar los números 1,2,3,4,5,6,7,8,9 en un cuadrado 3x3 de forma que todas las líneas (filas, columnas y diagonales) sumen igual.

A	B	C
D	E	F
G	H	I

```
cuadrado_1([A,B,C,D,E,F,G,H,I]) :-
    permutación([1,2,3,4,5,6,7,8,9],
```

```
[A,B,C,D,E,F,G,H,I]),
A+B+C ::= 15, D+E+F ::= 15,
G+H+I ::= 15, A+D+G ::= 15,
B+E+H ::= 15, C+F+I ::= 15,
A+E+I ::= 15, C+E+G ::= 15.
```

- Cálculo de soluciones:

```
?- cuadrado_1(L).
L = [6, 1, 8, 7, 5, 3, 2, 9, 4] ;
L = [8, 1, 6, 3, 5, 7, 4, 9, 2]
Yes
```

- Cálculo del número soluciones:

```
?- findall(_X,cuadrado_1(_X),_L),length(_L,N).
N = 8
Yes
```

### Cuadrado mágico por comprobaciones parciales

- Programa 2:

```
cuadrado_2([A,B,C,D,E,F,G,H,I]) :-
  select(A,[1,2,3,4,5,6,7,8,9],L1),
  select(B,L1,L2),
  select(C,L2,L3), A+B+C ::= 15,
  select(D,L3,L4),
  select(G,L4,L5), A+D+G ::= 15,
  select(E,L5,L6), C+E+G ::= 15,
  select(I,L6,L7), A+E+I ::= 15,
  select(F,L7,[H]), C+F+I ::= 15, D+E+F ::= 15.
```

- Cálculo de soluciones:

```
?- cuadrado_2(L).
L = [2, 7, 6, 9, 5, 1, 4, 3, 8] ;
L = [2, 9, 4, 7, 5, 3, 6, 1, 8]
Yes
```

- Comprobación que las dos definiciones dan las *mismas* soluciones.

```
?- setof(_X,cuadrado_1(_X),_L),
   setof(_X,cuadrado_2(_X),_L).
Yes
```



## Comparación de eficiencia del cuadrado mágico

```
?- time(cuadrado_1(_X)).
161,691 inferences in 0.58 seconds

?- time(cuadrado_2(_X)).
  1,097 inferences in 0.01 seconds

?- time(setof(_X,cuadrado_1(_X),_L)).
812,417 inferences in 2.90 seconds

?- time(setof(_X,cuadrado_2(_X),_L)).
  7,169 inferences in 0.02 seconds
```

## 2.3. Uso de listas o términos compuestos

### Uso de listas o términos compuestos

- Ejemplos:

```
?- setof(N,between(1,1000,N),L1),
   asserta(con_lista(L1)),
   Term =.. [f|L1],
   asserta(con_term(Term)).
?- listing(con_lista).
con_lista([1, 2, ..., 999, 1000]).
?- listing(con_term).
con_term(f(1, 2,...,999, 1000)).
?- time((con_lista(_L),member(1000,_L))).
1,001 inferences in 0.00 seconds
?- time((con_term(_T),arg(_,_T,1000))).
2 inferences in 0.00 seconds
```

- En general, sólo usar listas cuando el número de argumentos no es fijo o es desconocido.

## 2.4. Uso de la unificación implícita

### Uso de la unificación implícita

- `intercambia(+T1,-T2)` se verifica si `T1` es un término con dos argumentos y `T2` es un término con el mismo símbolo de función que `T1` pero sus argumentos intercambiados. Por ejemplo,

```
?- intercambia(opuesto(3,-3),T).
T = opuesto(-3, 3)
```

```
intercambia_1(T1, T2) :-
    functor(T1,F,2),    functor(T2,F,2),
    arg(1,T1,X1),      arg(2,T1,Y1),
    arg(1,T2,X2),      arg(2,T2,Y2),
    X1 = Y2,           X2 = Y1.

intercambia_2(T1,T2) :-
    T1 =.. [F,X,Y],
    T2 =.. [F,Y,X].
```

- `lista_de_tres(L)` se verifica si `L` es una lista de 3 elementos.
- Definición 1:

```
lista_de_tres(L) :-
    length(L, N),
    N = 3.
```

- Definición 2 (sin unificación explícita):

```
lista_de_tres(L) :-
    length(L,3).
```

- Definición 3 (sin `length`):

```
lista_de_tres([_,_,_]).
```

## 2.5. Acumuladores

### Acumuladores

- `inversa(+L1,-L2)`, `reverse(L1,L2)`, se verifica si `L2` es la lista inversa de `L1`. Por ejemplo,

```
?- inversa([a,b,c],L).
L = [c, b, a]
```

- Definición de `inversa` con `append` (no recursiva final):

```

inversa_1([], []).
inversa_1([X|L1], L2) :-
    inversa_1(L1, L3),
    append(L3, [X], L2).

```

- Definición de inversa con acumuladores (recursiva final):

```

inversa_2(L1, L2) :-
    inversa_2_aux(L1, [], L2).

inversa_2_aux([], L, L).
inversa_2_aux([X|L], Acum, L2) :-
    inversa_2_aux(L, [X|Acum], L2).

```

## Comparación de eficiencia

```

?- findall(_N, between(1, 1000, _N), _L1),
   time(inversa_1(_L1, _)), time(inversa_2(_L1, _)).
501,501 inferences in 0.40 seconds
1,002 inferences in 0.00 seconds

?- findall(_N, between(1, 2000, _N), _L1),
   time(inversa_1(_L1, _)), time(inversa_2(_L1, _)).
2,003,001 inferences in 1.59 seconds
2,002 inferences in 0.00 seconds

?- findall(_N, between(1, 4000, _N), _L1),
   time(inversa_1(_L1, _)), time(inversa_2(_L1, _)).
8,006,001 inferences in 8.07 seconds
4,002 inferences in 0.02 seconds

```

## 2.6. Uso de lemas

### Uso de lemas

- fibonacci(N, X) se verifica si X es el N-ésimo término de la sucesión de Fibonacci, definida por
 
$$f(1) = 1$$

$$f(2) = 1$$

$$f(n) = f(n-1) + f(n-2), \text{ si } n > 2$$

```

fibonacci_1(1,1).
fibonacci_1(2,1).
fibonacci_1(N,F) :-
    N > 2,
    N1 is N-1,
    fibonacci_1(N1,F1),
    N2 is N-2,
    fibonacci_1(N2,F2),
    F is F1 + F2.

```

■ Definición de Fibonacci con lemas

```

:- dynamic fibonacci_2/2.

fibonacci_2(1,1).
fibonacci_2(2,1).
fibonacci_2(N,F) :-
    N > 2,
    N1 is N-1,
    fibonacci_2(N1,F1),
    N2 is N-2,
    fibonacci_2(N2,F2),
    F is F1 + F2,
    asserta(fibonacci_2(N,F)).

```

### Comparación de eficiencia en el uso de lemas

```

?- time(fibonacci_1(20,N)).
40,585 inferences in 1.68 seconds
N = 6765

```

```

?- time(fibonacci_2(20,N)).
127 inferences in 0.00 seconds
N = 6765

```

```

?- listing(fibonacci_2).
fibonacci_2(20, 6765). fibonacci_2(19, 4181). ...

```

```

?- time(fibonacci_2(20,N)).
3 inferences in 0.00 seconds
N = 6765

```

## Uso de lemas y acumuladores

- Definición de Fibonacci con un acumulador

```
fibonacci_3(N,F) :-
    fibonacci_3_aux(N,_,F).
```

`fibonacci_3_aux(+N, -F1, -F2)` se verifica si `F1` es fibonacci de `N-1` y `F2` es fibonacci de `N`.

```
fibonacci_3_aux(0,_,0).
fibonacci_3_aux(1,0,1).
fibonacci_3_aux(N,F1,F) :-
    N > 1,
    N1 is N-1,
    fibonacci_3_aux(N1,F2,F1),
    F is F1 + F2.
```

## Comparación de eficiencia en el uso de lemas y acumuladores

- Comparación;

```
?- time(fibonacci_1(20,N)).
40,585 inferences in 1.68 seconds

?- time(fibonacci_2(20,N)).
127 inferences in 0.00 seconds

?- time(fibonacci_3(20,N)).
21 inferences in 0.00 seconds
```

## 2.7. Determinismo

### Determinismo

- `descompone(+E, -N1, -N2)` se verifica si `N1` y `N2` son dos enteros no negativos tales que `N1+N2=E`.
- Definición no determinista:

```
descompone_1(E, N1, N2) :-
    between(0, E, N1),
    between(0, E, N2),
    E ::= N1 + N2.
```

- Definición determinista:

```
descompone_2(E, N1, N2) :-
    between(0, E, N1),
    N2 is E - N1.
```

- Comparación de eficiencia:

```
?- time(setof(_N1+_N2,descompone_1(1000,_N1,_N2),_L)).
1,004,019 inferences in 1.29 seconds

?- time(setof(_N1+_N2,descompone_2(1000,_N1,_N2),_L)).
2,018 inferences in 0.01 seconds
```

## 2.8. Añadir al principio

### Añadir al principio

- `lista_de_cuadrados(+N, ?L)` se verifica si `L` es la lista de los cuadrados de los números de 1 a `N`. Por ejemplo,

```
?- lista_de_cuadrados(5,L).
L = [1, 4, 9, 16, 25]
```

- Programa 1 (añadiendo por detrás):

```
lista_de_cuadrados_1(1,[1]).
lista_de_cuadrados_1(N,L) :-
    N > 1,
    N1 is N-1,
    lista_de_cuadrados_1(N1,L1),
    M is N*N,
    append(L1,[M],L).
```

- Programa 2 (añadiendo por delante):

```
lista_de_cuadrados_2(N,L) :-
    lista_de_cuadrados_2_aux(N,L1),
    reverse(L1,L).

lista_de_cuadrados_2_aux(1,[1]).
lista_de_cuadrados_2_aux(N,[M|L]) :-
    N > 1,
    M is N*N,
```

```
N1 is N-1,
lista_de_cuadrados_2_aux(N1,L).
```

- Programa 3 (con findall):

```
lista_de_cuadrados_3(N,L) :-
  findall(M,(between(1,N,X), M is X*X),L).
```

- Comparación:

```
?- time(lista_de_cuadrados_1(10000,_L)).
50,044,996 inferences in 24.34 seconds
Yes

?- time(lista_de_cuadrados_2(10000,_L)).
50,000 inferences in 0.06 seconds
Yes

?- time(lista_de_cuadrados_3(10000,_L)).
20,012 inferences in 0.04 seconds
Yes
```

## 2.9. Listas de diferencias

### Listas de diferencias

- Representaciones de  $[a, b, c]$  como listas de diferencias:

```
[a,b,c,d] - [d]
[a,b,c,1,2,3] - [1,2,3]
[a,b,c|X] - X
[a,b,c] - []
```

- `conc_ld(LD1,LD2,LD3)` se verifica si `LD3` es la lista de diferencia correspondiente a la concatenación de las listas de diferencia `LD1` y `LD2`. Por ejemplo,

```
?- conc_ld([a,b|RX]-RX,[c,d|RY]-RY,Z-[]).
RX = [c, d]   RY = []   Z = [a, b, c, d]
?- conc_ld([a,b|_RX]-_RX,[c,d|_RY]-_RY,Z-[]).
Z = [a, b, c, d]
```

```
conc_ld(A-B,B-C,A-C).
```

## Bibliografía

### Bibliografía

1. I. Bratko *Prolog Programming for Artificial Intelligence (2nd ed.)* (Addison–Wesley, 1990)
  - Cap. 8: “Programming Style and Technique”
2. W.F. Clocksin y C.S. Mellish *Programming in Prolog (Fourth Edition)* (Springer Verlag, 1994)
  - Cap. 6: “Using Data Structures”
3. M.A. Covington *Efficient Prolog: A Practical Guide*



## **Capítulo 12**

### **Aplicaciones de PD: Problemas de grafos y de las reinas**

# Programación declarativa (2008–09)

Tema 12: Problemas de grafos y de las reinas

José A. Alonso Jiménez

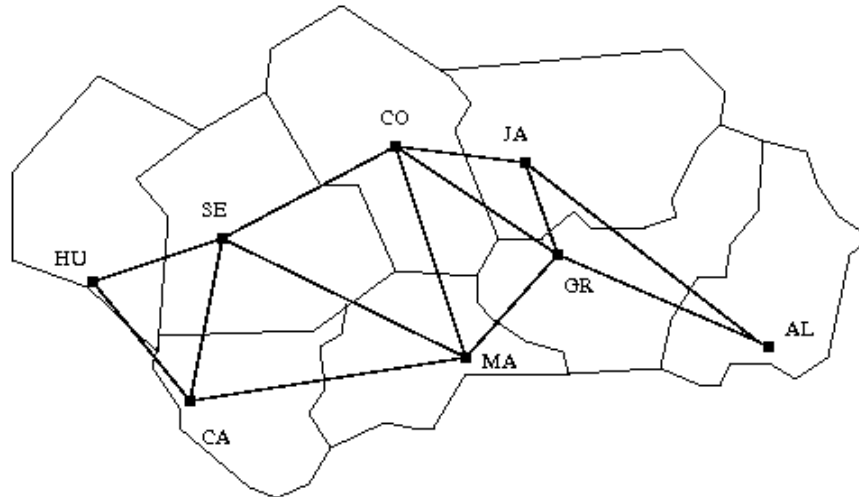
## Índice

<b>1. Problemas de grafos</b>	<b>1</b>
1.1. Representación de grafos . . . . .	1
1.2. Caminos en un grafo . . . . .	2
1.3. Caminos hamiltonianos en un grafo . . . . .	2
<b>2. El problema de las reinas</b>	<b>3</b>
2.1. Especificación del problema de las reinas . . . . .	3
2.2. Representación mediante filas y columnas . . . . .	3
2.3. Representación mediante columnas . . . . .	4
2.4. Representación mediante filas, columnas y diagonales . . . . .	5

# 1. Problemas de grafos

## 1.1. Representación de grafos

### Grafo de Andalucía



### Representación del grafo

- `arcos(+L)` se verifica si `L` es la lista de arcos del grafo.

```
arcos([huelva-sevilla, huelva-cádiz,
      cádiz-sevilla, sevilla-málaga,
      sevilla-córdoba, Córdoba-málaga,
      Córdoba-granada, Córdoba-jaén,
      jaén-granada, jaén-almería,
      granada-almería]).
```

### Adyacencia y nodos

- `adyacente(?X,?Y)` se verifica si `X` e `Y` son adyacentes.

```
adyacente(X,Y) :-
  arcos(L),
  (member(X-Y,L) ; member(Y-X,L)).
```

- `nodos(?L)` se verifica si `L` es la lista de nodos.

```
nodos(L) :-
  setof(X,Y^adyacente(X,Y),L).
```

## 1.2. Caminos en un grafo

### Camino

- `camino(+A,+Z,-C)` se verifica si `C` es un camino en el grafo desde el nodo `A` al `Z`. Por ejemplo,

```
?- camino(sevilla,granada,C).
C = [sevilla, córdoba, granada] ;
C = [sevilla, Málaga, córdoba, granada]
Yes
```

```
camino(A,Z,C) :-
    camino_aux(A,[Z],C).
```

- `camino_aux(+A,+CP,-C)` se verifica si `C` es una camino en el grafo compuesto de un camino desde `A` hasta el primer elemento del camino parcial `CP` (con nodos distintos a los de `CP`) junto `CP`.

```
camino_aux(A,[A|C1],[A|C1]).
camino_aux(A,[Y|C1],C) :-
    adyacente(X,Y),
    not(member(X,[Y|C1])),
    camino_aux(A,[X,Y|C1],C).
```

## 1.3. Caminos hamiltonianos en un grafo

### Camino hamiltoniano

- `hamiltoniano(-C)` se verifica si `C` es un camino hamiltoniano en el grafo (es decir, es un camino en el grafo que pasa por todos sus nodos una vez). Por ejemplo,

```
?- hamiltoniano(C).
C = [almería, jaén, granada, córdoba, Málaga, sevilla, huelva, cádiz]
?- findall(_C,hamiltoniano(_C),_L), length(_L,N).
N = 16
```

- Primera definición de hamiltoniano

```
hamiltoniano_1(C) :-
    camino(_,_ ,C),
    nodos(L),
    length(L,N),
    length(C,N).
```

- Segunda definición de hamiltoniano

```
hamiltoniano_2(C) :-
    nodos(L),
    length(L,N),
    length(C,N),
    camino(_,_ ,C).
```

- Comparación de eficiencia

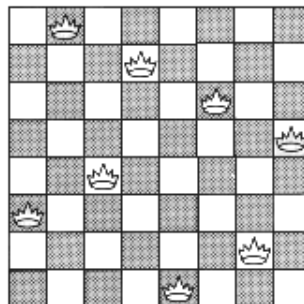
```
?- time(findall(_C,hamiltoniano_1(_C),_L)).
37,033 inferences in 0.03 seconds (1234433 Lips)
?- time(findall(_C,hamiltoniano_2(_C),_L)).
13,030 inferences in 0.01 seconds (1303000 Lips)
```

## 2. El problema de las reinas

### 2.1. Especificación del problema de las reinas

#### El problema de las 8 reinas

- El problema de las ocho reinas consiste en colocar 8 reinas en un tablero rectangular de dimensiones 8 por 8 de forma que no se encuentren más de una en la misma línea: horizontal, vertical o diagonal.



### 2.2. Representación mediante filas y columnas

#### Representación mediante filas y columnas

- Sesión:

```
?- tablero(S), solución(S).
S = [1-4, 2-2, 3-7, 4-3, 5-6, 6-8, 7-5, 8-1] ;
```

```
S = [1-5, 2-2, 3-4, 4-7, 5-3, 6-8, 7-6, 8-1] ;
S = [1-3, 2-5, 3-2, 4-8, 5-6, 6-4, 7-7, 8-1]
Yes
```

- `tablero(L)` se verifica si `L` es una lista de posiciones que representan las coordenadas de 8 reinas en el tablero.

```
tablero(L) :-
    findall(X-Y,between(1,8,X),L).
```

- `solución_1(?L)` se verifica si `L` es una lista de pares de números que representan las coordenadas de una solución del problema de las 8 reinas.

```
solución_1([]).
solución_1([X-Y|L]) :-
    solución_1(L),
    member(Y,[1,2,3,4,5,6,7,8]),
    no_ataca(X-Y,L).
```

- `no_ataca([X,Y],L)` se verifica si la reina en la posición  $(X, Y)$  no ataca a las reinas colocadas en las posiciones correspondientes a los elementos de la lista `L`.

```
no_ataca(_, []).
no_ataca(X-Y,[X1-Y1|L]) :-
    X =\= X1,      Y =\= Y1,
    X-X1 =\= Y-Y1, X-X1 =\= Y1-Y,
    no_ataca(X-Y,L).
```

## 2.3. Representación mediante columnas

### Representación mediante columnas

- `solución_2(L)` se verifica si `L` es una lista de 8 números,  $[n_1, \dots, n_8]$ , de forma que si las reinas se colocan en las casillas  $(1, n_1), \dots, (8, n_8)$ , entonces no se atacan entre sí.

```
solución_2(L) :-
    permutación([1,2,3,4,5,6,7,8],L),
    segura(L).
```

- `segura(L)` se verifica si `L` es una lista de  $m$  números  $[n_1, \dots, n_m]$  tal que las reinas colocadas en las posiciones  $(x, n_1), \dots, (x + m, n_m)$  no se atacan entre sí.

```
segura([]).
segura([X|L]) :-
    segura(L),
    no_ataca(X,L,1).
```

- `no_ataca(Y,L,D)` se verifica si  $Y$  es un número,  $L$  es una lista de números  $[n_1, \dots, n_m]$  y  $D$  es un número tales que las reinas colocada en la posición  $(X, Y)$  no ataca a las colocadas en las posiciones  $(X + D, n_1), \dots, (X + D + m, n_m)$ .

```
no_ataca(_, [], _).
no_ataca(Y, [Y1|L], D) :-
    Y1-Y =\= D,
    Y-Y1 =\= D,
    D1 is D+1,
    no_ataca(Y,L,D1).
```

## 2.4. Representación mediante filas, columnas y diagonales

### Representación mediante filas, columnas y diagonales

- `solución_3(?L)` se verifica si  $L$  es una lista de 8 números,  $[n_1, \dots, n_8]$ , de forma que si las reinas se colocan en las casillas  $(1, n_1), \dots, (8, n_8)$ , entonces no se atacan entre sí.

```
solución_3(L) :-
    solución_3_aux(
        L,
        [1,2,3,4,5,6,7,8],
        [1,2,3,4,5,6,7,8],
        [-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7],
        [2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]).
```

- `solución_3_aux(?L,+Dx,+Dy,+Du,+Dv)` se verifica si  $L$  es una permutación de los elementos de  $Dy$  de forma que si  $L$  es  $[y_1, \dots, y_n]$  y  $Dx$  es  $[1, \dots, n]$ , entonces  $y_j - j$  ( $1 \leq j \leq n$ ) son elementos distintos de  $Du$  e  $y_j + j$  ( $1 \leq j \leq n$ ) son elementos distintos de  $Dv$ .

```
solucion_aux([], [], _Dy, _Du, _Dv).
solucion_aux([Y|Ys], [X|Dx1], Dy, Du, Dv) :-
    select(Y,Dy,Dy1),
    U is X-Y,
    select(U,Du,Du1),
```

```
V is X+Y,
select(V,Dv,Dv1),
solucion_aux(Ys,Dx1,Dy1,Du1,Dv1).
```

### Comparaciones de eficiencia

```
?- time((findall(_S,(tablero_1(_S),solucion_1(_S)),_L),length(_L,N))).
211,330 inferences in 0.12 seconds (1761083 Lips)
N = 92

?- time((findall(_S,solucion_2(_S),_L),length(_L,N))).
1,422,301 inferences in 0.72 seconds (1975418 Lips)
N = 92

?- time((findall(_S,solucion_3(_S),_L),length(_L,N))).
120,542 inferences in 0.07 seconds (1722029 Lips)
N = 92
```

### Búsqueda de todas las soluciones para N reinas:

N	solución 1		solución 2		solución 3	
	inferencias	seg.	inferencias	seg.	inferencias	seg.
4	401	0.00	543	0.00	546	0.00
6	8,342	0.00	20,844	0.01	6,660	0.01
8	195,628	0.15	1,422,318	0.91	120,614	0.09
10	5,303,845	4.05	150,300,540	96.82	2,774,095	2.01
12	182,574,715	147.22			83,067,721	64.93

## Bibliografía

### Bibliografía

1. I. Bratko *Prolog Programming for Artificial Intelligence (2nd ed.)* (Addison–Wesley, 1990)
  - Cap. 4: “Using Structures: Example Programs”
  - Cap. 9: “Operations on Data Structures”
2. L. Sterling y E. Shapiro *The Art of Prolog (2nd edition)* (The MIT Press, 1994)
  - Cap. 2 “Database programming”



# Bibliografía

- [1] J. A. Alonso *Introducción a la programación lógica con Prolog*.  
[http://www.cs.us.es/~jalonso/publicaciones/2006-int\\_prolog.pdf](http://www.cs.us.es/~jalonso/publicaciones/2006-int_prolog.pdf)
- [2] P. Blackburn, J. Bos y K. Striegnitz *Learn Prolog Now!*.  
<http://www.coli.uni-sb.de/~kris/learn-prolog-now>
- [3] I. Bratko *Prolog Programming for Artificial Intelligence (3 ed.)* (Addison–Wesley, 2001).
- [4] W. F. Clocksin y C.S. Mellish *Programming in Prolog (Fourth Edition)*. (Springer Verlag, 1994).
- [5] M. A. Covington *Efficient Prolog: A Practical Guide*.  
<http://www.ai.uga.edu/ftplib/ai-reports/ai198908.pdf>
- [6] M. A. Covington, D. Nute y A. Vellino *Prolog Programming in Depth*. (Prentice Hall, 1997).
- [7] H. C. Cunningham *Notes on Functional Programming with Haskell*. (Technical Report, University of Mississippi, 2007).
- [8] P. Flach *Simply Logical (Intelligent Reasoning by Example)*. (John Wiley, 1994).
- [9] J. Fokker *Programación funcional*. (Technical Report, Universidad de Utrech, 1996).
- [10] U. Nilsson y J. Maluszynski *Logic, Programming and Prolog (2nd ed.)*  
<http://www.ida.liu.se/~ulfni/lpp>
- [11] B.C. Ruiz, F. Gutiérrez, P. Guerrero, y J- Gallardo *Razonando con Haskell (Un curso sobre programación funcional)*. (Thompson, 2004).
- [12] L. Sterling y E. Shapiro *The Art of Prolog*. (MIT Press, 1994).
- [13] S. Thompson *Haskell: The Craft of Functional Programming*. (Addison–Wesley, second edition, 1999).
- [14] T. Van Le *Techniques of Prolog Programming*. (John Wiley, 1993).

- [15] E. P. Wentworth *Introduction to Funcional Programming*. (Technical Report, Rhodes University, 1994).