Programación declarativa (2007–08)

Tema 1: Programación funcional básica

José A. Alonso Jiménez

Grupo de Lógica Computacional
Departamento de Ciencias de la Computación e I.A.
Universidad de Sevilla

Tema 1: Programación funcional básica (I)

Tema 1: Programación funcional básica (II)

4. Definición de funciones

- Definición por combinación
- Definición por distinción de casos
- Definición por análisis de patrones
- Definición por recursión
- Indentación y comentarios

5. Tipado

- Clases de errores
- Expresiones de tipo
- Polimorfismo
- Sobrecarga

Tema 1: Programación funcional básica

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).

Tema 1: Programación funcional básica

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.

Tema 1: Programación funcional básica

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> hughs
__   __ __  __  ____   ___      ------------------------------------------
||   || ||  || ||  || ||__      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/hug
||   || Version: September 2006 ------------------------------------------

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

Type :? for help
Hugs> 2+3
5
```

Inicio, cálculo, ayuda y fin de Hugs

```
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
```

Inicio, cálculo, ayuda y fin de Hugs

```
: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
```

Ejemplos de evaluaciones

- 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
```

Tema 1: Programación funcional básica

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
```

Tema 1: Programación funcional básica

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
  `: # % & * + - = . / \ < > ? ! @ ^ |`

Tema 1: Programación funcional básica

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`	logaritmno natural
`exp`	exponencial

- Funciones predefinidas de transfomación entre enteros y reales:

`FromInteger`	de entero a real
`round`	redondea un real a entero

Tema 1: Programación funcional básica

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
```

Tema 1: Programación funcional básica

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
```

Funciones predefinidas sobre listas

- `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]
```

Tema 1: Programación funcional básica

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]
```

Tema 1: Programación funcional básica

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)
```

Definiciones por combinación con un parámetro

- `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 conjuntos de `n` elementos. Por ejemplo,

  `n_combinaciones 6 2` $\rightsquigarrow$ `15`

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

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

  `raices 1 3 2` $\rightsquigarrow$ `[-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)
```

Tema 1: Programación funcional básica

Definiciones por distinción de casos

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

```
abs (-3) ⇝ 3
```

Prelude

```
abs x | x < 0  = -x
      | x >= 0 = x
```

- `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

```
signum x | x > 0      = 1
         | x == 0     = 0
         | otherwise  = -1
```

Definiciones por distinción de casos

- 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.

Tema 1: Programación funcional básica

Clases de parámetros

- Parámetros formales: los usados en las definiciones de funciones.
- Parámetos 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 l` es la suma de los dos elementos de `l` si `l` 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]
```

Definiciones con patrones

- `suma l` es la suma de los elementos de la lista `l` cuando `l` 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 preludio

- `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
```

Tema 1: Programación funcional básica

Ejemplos de definiciones por recursión numérica

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

  `fac 20` $\rightsquigarrow$ `2432902008176640000`

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

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

  `potencia 2 4` $\rightsquigarrow$ `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]
```

Tema 1: Programación funcional básica

Indentación y comentarios

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

Tema 1: Programación funcional básica

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 defi
```

- 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

Tema 1: Programación funcional básica

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.

Tema 1: Programación funcional básica

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 polimosrfismo con listas:

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

- Ejemplo de polimosrfismo 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
```

Tema 1: Programación funcional básica

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 -> a -> 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*.