

Estructuras de Datos.

Grados en Ingeniería Informática, del Software y de Computadores

ETSI Informática

Universidad de Málaga

1. Introducción a la Programación Funcional

José E. Gallardo, Francisco Gutiérrez, Pablo López

Dpto. Lenguajes y Ciencias de la Computación

Universidad de Málaga

Índice

- Programación Funcional
 - Programación Imperativa vs. Programación Funcional
 - Funciones puras e impuras
 - Ventajas de la Programación Funcional
- Introducción a Haskell
 - Identificadores
 - Tipos básicos
 - Definiciones de funciones. Aplicaciones
 - Operadores, precedencia y asociatividad
 - Evaluación de expresiones
 - Tuplas
 - Polimorfismo y sobrecarga
 - Ecuaciones condicionales y expresiones
 - Funciones definidas parcialmente
 - Definiciones locales
 - Operadores vs. Funciones
 - Introducción a QuickCheck

Programación imperativa

- Los cómputos manipulan variables **mutables**

http://en.wikipedia.org/wiki/Imperative_programming

- Estas variables mutables representan la memoria o el estado del computador
- Los algoritmos modifican las variables hasta obtener el resultado deseado

```
int factorial(int n) {  
    int p = 1;  
    for(int i = 1; i <= n; i++)  
        p = p * i;  
    return p;  
}
```

Las variables i y p son alteradas durante la ejecución

Programación funcional pura

- Un estilo de programación basado en cómputos a través de funciones **matemáticas** (o puras) :

http://en.wikipedia.org/wiki/Functional_programming

- Un programa es una función
 - Toma datos a través de sus **argumentos**
 - Realiza cálculos con estos datos a través de **expresiones**
 - Devuelve nuevos datos (**resultados** del cómputo)



- ¡ Nada cambia !
 - No existe el concepto de variable mutable típico de la programación imperativa

Factorial en Haskell

- Función para calcular factorial en Haskell:

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

- Evaluación paso a paso:

```
factorial 3
=> {- ya que 3 > 0 -}
    3 * factorial 2
=> {- ya que 2 > 0 -}
    3 * (2 * factorial 1)
=> {- ya que 1 > 0 -}
    3 * (2 * (1 * factorial 0))
=> {- base case -}
    3 * (2 * (1 * 1))
=> {- aritmética -}
    ...
6
```

No hay variables mutables. Tan solo funciones y expresiones

Funciones matemáticas (puras)

- Una función matemática **siempre** devuelve el mismo resultado para los mismos valores de sus argumentos:

http://en.wikipedia.org/wiki/Pure_function

$$x = y \Rightarrow f(x) = f(y) \quad (\text{Regla de Leibniz})$$

- Esto conduce a la propiedad de **transparencia referencial**:

Una misma expresión denota siempre el mismo valor, sea cual sea el punto del programa o el contexto en que aparezca.

- Simplifica el razonamiento sobre las propiedades de los programas (en particular, la corrección)
- Podemos ‘calcular’ con programas al igual que en matemáticas ‘calculamos’ con expresiones

Funciones matemáticas (puras) (II)

- Un ejemplo de función pura:

```
int nice(int n) {  
    return n * n + 10;  
}
```

- **nice(5) siempre** devuelve 35
- Si $x = y$ entonces $\text{nice}(x) = \text{nice}(y)$ ☺

Funciones impuras

- La siguiente no es una función pura:

```
int state = 0;

int weird(int n) {
    state += 1;
    return n + state;
}
```

- La primera llamada `weird(5)` devuelve 6
- La segunda llamada `weird(5)` devuelve 7
- Para el mismo valor del argumento `weird` devuelve diferentes resultados ☹

Funciones impuras (II)

- La siguiente **tampoco** es una función pura (no es matemática):

```
int getchar(void);
```

- Cada llamada puede devolver un valor distinto

La programación con Lenguajes Funcionales

- La programación funcional consiste en escribir programas (funciones) combinando otras funciones
- En los **lenguajes funcionales puros**:
 - Solamente podemos definir funciones puras
 - Se satisface la transparencia referencial 😊
Miranda, Clean, **Haskell**, ...
- En los **lenguajes funcionales impuros**:
 - Se permite definir funciones impuras 😞
ML, F#, Scheme, ...

Ventajas de la programación funcional pura

- **Optimización**: el compilador puede realizar transformaciones sofisticadas con objeto de optimizar el comportamiento
- **Paralelismo**: partes de una expresión pueden evaluarse en cualquier orden sin alterar el resultado; incluso varios procesadores pueden intervenir en la evaluación de la misma expresión.
 - La Programación Funcional Pura es intrínsecamente paralela 😊
 - La Programación Imperativa es intrínsecamente secuencial 😞
- **Memoization**: los resultados producidos por las llamadas a una función pueden ser reutilizados (recordados) en sucesivas llamadas con los mismos valores de los argumentos

Un ejemplo sencillo de optimización

- En un LFP (lenguaje funcional puro), la definición:

```
g :: Integer -> Integer  
g x = f x + h (10*x) + f x
```

puede reemplazarse por :

```
g :: Integer -> Integer  
g x = 2 * f x + h (10*x)
```

sin modificar el significado del programa.

- Esta sencilla transformación es una **optimización**:
 $f x$ solamente se computa (evalúa) una vez
- Además, la expresión $2 * f x + h (10*x)$ puede evaluarse vía paralelismo
- Lo anterior no es válido en lenguajes impuros ☹

Haskell

- Un Lenguaje Funcional Puro estándar



- Principales características:
 - Pureza funcional
 - Tipificación estática
 - Perezoso (Lazy)

Identificadores

- Haskell distingue mayúsculas de minúsculas (case sensitive) !
- Los identificadores (nombres) de **funciones** y **argumentos**:
 - Deben empezar con una letra minúscula o el carácter `_` (*underscore*) y éste puede ir seguido de letras (mayúsculas o minúsculas), dígitos, tildes (`'`) o `_`
 - Ejemplos válidos de identificadores:
`f f' f2 x _uno factorial árbol toList p1_gcd`
 - Ejemplos no válidos:
`Function 2f f!y`
- Los nombres de **tipos** deben empezar con una letra mayúscula `Int, Integer, Bool, Float, Double`, etc.
- Los nombres de **operadores**:
 - Pueden contener uno o varios símbolos
 - El primero **no** puede ser dos-puntos (`:`)
 - Ejemplos válidos:
`+ * - !! <= ==> +:`

Tipos numéricicos

- **Int** un subconjunto acotado de los números enteros
 - operaciones rápidas; posibilidad de *overflow* o *underflow*

Los ejemplos asumen una arquitectura de 32 bits

```
Prelude> maxBound :: Int  
2147483647
```

$2^{31} - 1$

```
Prelude> (maxBound :: Int) + 1  
-2147483648
```

```
Prelude> minBound :: Int  
-2147483648
```

```
Prelude> (1000000000 :: Int) * 1000000000  
-1981284352
```

Tipos numéricos (II)

- **Integer** el conjunto ‘completo’ de los enteros

```
Prelude> (10000000000 :: Integer) * 1000000000  
10000000000000000000
```

- **Float** subconjunto de los reales en simple precisión

2 -3.5 1.5E10

```
Prelude> (1e-45 :: Float)  
1.0e-45
```

```
Prelude> (1e-45 :: Float) / 2  
0.0
```

- Double subconjunto de los reales en doble precisión

Prelude> (1e-300 :: Double) / 2

5.0e-301

Booleanos

- Bool tiene solamente dos valores: True y False

- conjunción lógica (and):

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

- disyunción lógica (or):

`(||) :: Bool -> Bool -> Bool`

- negación lógica:

`not :: Bool -> Bool`



http://en.wikipedia.org/wiki/George_Boole

`(1 < 3) && (x >= y) || not (primo y)`

Hay evaluación en cortocircuito

Caracteres

- Char conjunto de caracteres Unicode :

'a', 'b', ..., 'z', 'A', 'B', ..., 'Z',
'0', '1', ..., '9', '+', '?', ...

- Muchas funciones sobre caracteres aparecen definidas en el módulo o librería Data.Char:

```
import Data.Char
```

```
g, g' :: Char -> Char
```

```
g x = chr (ord x + ord 'A' - ord 'a')
```

```
g' x = toUpper x
```

Parcialización (currying)

- Haskell usa notación **parcializada (curried)**

<http://en.wikipedia.org/wiki/Currying>

- Introducida por
Moses Schönfinkel (1889-1942)
y popularizada por
Haskell Curry (1900-1982)



- Para aplicar una función a sus argumentos:
 - Escribimos el identificador de la función y a continuación los argumentos separados por **espacios**
 - El uso de **paréntesis** sólo será necesario para agrupar términos compuestos y modificar el efecto de las prioridades de las operaciones

Definición de funciones

toma un entero

devuelve un entero

twice :: Integer -> Integer
twice x = x + x

nombre de la función

argumento

resultado

toma dos enteros
y devuelve un entero

square :: Integer -> Integer
square x = x * x

pythagoras :: Integer -> Integer -> Integer
pythagoras x y = square x + square y

Definición de funciones (II)

```
square :: Integer -> Integer  
square x = x*x
```

```
pythagoras :: Integer -> Integer -> Integer  
pythagoras x y = square x + square y
```

square sólo se aplica al 5

square 10 → 10*10 → 100

square se aplica a la suma 5 + 1.

Los paréntesis son necesarios

square 5 + 1 → 5*5 + 1 → ... → 26

square (5 + 1) → square 6 → ... → 36

pythagoras se aplica a los dos argumentos

pythagoras 10 6 → square 10 + square 6
→ ... → 136

El segundo argumento es compuesto y necesita paréntesis

pythagoras 10 (twice 3)
→ square 10 + square (twice 3)
→ ... → 136

Definición de funciones. Condicionales (III)

Una expresión condicional, como cualquier otra expresión, puede formar parte del cuerpo de una función:

```
maxInteger :: Integer -> Integer -> Integer  
maxInteger x y = if x >= y then x else y
```

```
Main> maxInteger 4 17  
17
```

Definición de funciones. Recursión (IV)

- El cuerpo de una función puede contener la función que define

```
fact :: Int -> Int
fact n = if n == 0 then 1 else n * fact (n-1) -- recursividad
```

- El uso del tipo **Int** (subconjunto finito de enteros) causará problemas:

fact 16 → 2004189184

fact 17 → -288522240

- El problema se resuelve tomando **Integer** (una abstracción del conjunto completo de los enteros)

```
factorial :: Integer -> Integer
factorial n = if n == 0 then 1 else n * factorial (n-1)
```

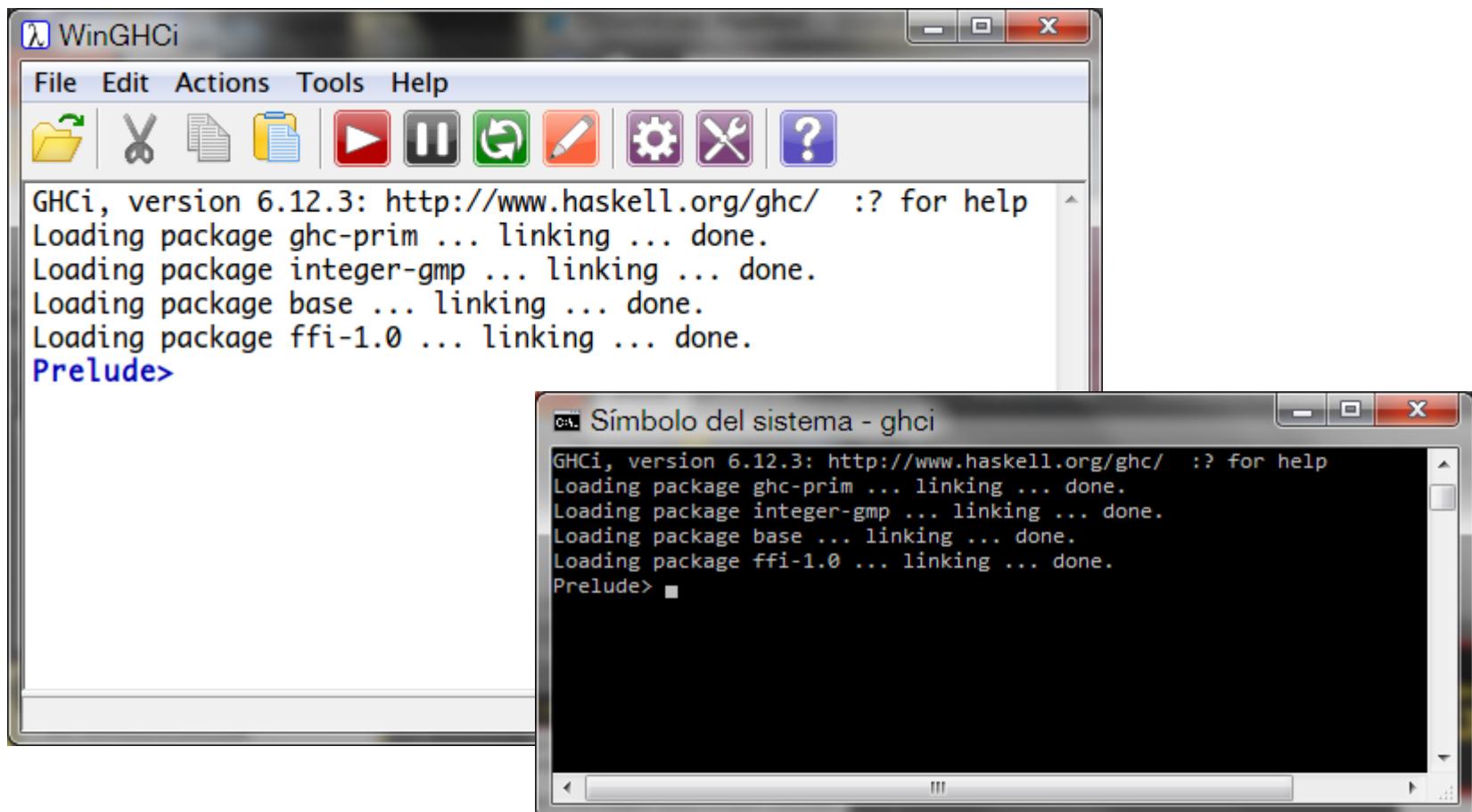
factorial 17 → 355687428096000

GHC y GHCi

- GHC es el *Glasgow Haskell Compiler*, un compilador de alta calidad para Haskell
- GHCi es el *Glasgow Haskell Compiler Interpreter*:
 - Con el uso del bucle *Read-Eval-Print* podemos evaluar programas Haskell
- GHC y GHCi están disponibles para las plataformas más populares: Windows, Linux y OSX
- En la página web de la asignatura aparecen instrucciones de instalación

GHC y GHCI (II)

■ Dos entornos para GHCI



Operadores

- El operador **unario** – es el único operador unario simbólico **prefijo**:
 - $-1 \quad -(1+2) \rightarrow -3$
- El resto de operadores simbólicos son **binarios** y se usan directamente en **forma infija**:
 - $1+2 \quad \rightarrow \quad 3$
 - $1 + 2*3 \quad \rightarrow \quad 1 + 6$
 - $(1+2)*3 \quad \rightarrow \quad 3*3$
 - $1 + -2 \quad \rightarrow \quad \text{Precedence parsing error}$
 - $1 + (-2) \quad \rightarrow \quad -1$

Nivel de precedencia

Nivel	Operador predefinido	
Máxima precedencia	10	Aplicación de una función
	9	. !!
	8	\wedge $\wedge\wedge$ $\ast\ast$
	7	\ast /
	6	+ -
	5	: ++
	4	$\ast\ast$ /= < \leq > \geq
	3	&&
	2	
	1	\gg $\gg\ast$ = \ll
Mínima precedencia	0	\$ \$!

La aplicación de una función tiene precedencia máxima

La multiplicación * tiene mayor nivel de precedencia que la suma +

Asociatividad

- Esta característica permite resolver la ambigüedad en ausencia de paréntesis:

- Si un operador \diamond asocia a la izquierda:

$$x_1 \diamond x_2 \diamond \dots x_n = (\dots((x_1 \diamond x_2) \diamond x_3) \dots \diamond x_n)$$

- Si \diamond asocia a la derecha:

$$x_1 \diamond x_2 \diamond \dots x_n = (x_1 \diamond \dots (x_{n-2} \diamond (x_{n-1} \diamond x_n)) \dots)$$

- Si \diamond no es asociativo:

$$x_1 \diamond x_2 \diamond \dots x_n$$



Error de compilación:
los paréntesis son
necesarios

Asociatividad (II)

Asociatividad	Operador
Derecha	<code>^ ^^ ** : && \$ \$!</code>
Izquierda	<code>* / + - >> >>= =<<</code>
No asociativo	<code>== /= < <= >= ></code>

Ejemplos:

$$10 - 3 - 3 \rightarrow ((10 - 3) - 3)$$

Usa paréntesis si
desconoces la
asociatividad

$$2 \wedge 3 \wedge 4 \rightarrow (2 \wedge (3 \wedge 4))$$

$$1 < 3 < 4$$



Expresión errónea

Asociatividad vs ‘propiedad asociativa’

- Son diferentes conceptos que no deben ser confundidos
- La asociatividad es un atributo sintáctico de un operador en un lenguaje de programación
- El operador \diamond verifica la propiedad asociativa en sentido matemático si satisface:
$$\forall x, y, z . \ (x \diamond y) \diamond z = x \diamond (y \diamond z)$$
- La propiedad asociativa también es interesante computacionalmente:
 - Para evaluar $x \diamond y \diamond z$ se parentizará de acuerdo con el menor coste computacional.

Operadores y notación prefija

- Toda función de dos argumentos cuyo identificador sea literal puede ser utilizada en **forma infija** *encerrándolo* entre acentos graves (`)

```
Prelude> max 10 2
```

notación prefija

```
10
```

```
Prelude> 10 `max` 2
```

forma infija

```
10
```

```
Prelude> 10 max 2
```

Error

```
TYPE ERROR ...
```



```
Prelude> max 10 (max 3 15)
```

```
15
```

```
Prelude> 10 `max` 3 `max` 15
```

```
15
```

Si la función es asociativa
no son necesarios los
paréntesis

Operadores y notación prefija (II)

- Recíprocamente, cualquier operador simbólico puede utilizarse en **forma prefija** *encerrándolo* entre paréntesis

```
Prelude> 10 + 2
```

forma infija

```
12
```

```
Prelude> (+) 10 2
```

forma prefija

```
12
```

```
Prelude> + 10 2
```

Error

```
SYNTAX ERROR
```



Evaluación de expresiones

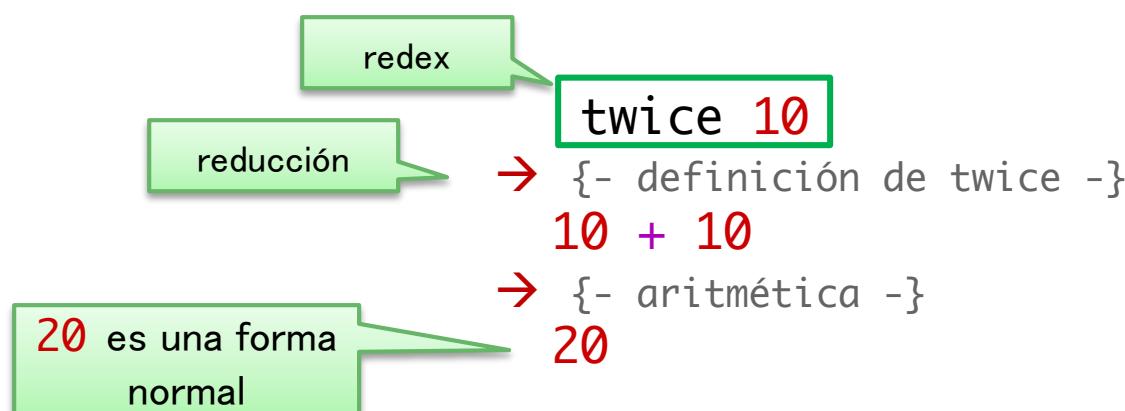
```
twice :: Integer -> Integer  
twice x = x + x
```

```
twice 10  
→ {- definición de twice -}  
10 + 10  
→ {- aritmética -}  
20
```



Evaluación de expresiones (II)

- Redex (*reducible expression*): parte de una expresión que puede ser simplificada
- Reducción: proceso de reescritura de un redex. Cada reducción es un paso de cómputo
- Forma Normal: expresión sin redexes



Evaluación de expresiones (III)

```
twice :: Integer -> Integer  
twice x = x + x
```

twice (10+2)

→ {- aritmética -}

twice 12

→ {- definición de twice -}

12 + 12

→ {- aritmética -}

24

Orden Aplicativo

Se reduce el redex más interno y más a la izquierda.

Los argumentos son evaluados antes que la función

Pros

Los argumentos compuestos se evalúan una sola vez

Contras

Pueden realizarse reducciones innecesarias.

Puede no conducir a la forma normal (**no es normalizante**)

Evaluación de expresiones (IV)

```
second :: Integer -> Integer -> Integer  
second x y = y
```

second (div 1 0) 2

→ {- aritmética -}

Exception: divide by zero

Orden Aplicativo

Se reduce el redex más interno y más a la izquierda.

Los argumentos son evaluados antes que la función

Pros

Los argumentos compuestos se evalúan una sola vez

Contras

Pueden realizarse reducciones innecesarias.

Puede no conducir a la forma normal (**no es normalizante**)

Evaluación de expresiones (V)

```
twice :: Integer -> Integer  
twice x = x + x
```

twice (10+2)
→ {- definición de twice -}
(10+2) + (10+2)
→ {- aritmética -}
12 + (10+2)
→ {- aritmética -}
12 + 12
→ {- aritmética -}
24

Orden Normal

Se reduce el redex más externo y más a la izquierda.

La función es aplicada antes de evaluar sus argumentos

Pros

Sólo se evalúan las expresiones necesarias.

Siempre conduce a la forma normal si ésta existe ([es normalizante](#))

Contras

Puede repetirse la evaluación de un argumento compuesto

Evaluación de expresiones (VI)

```
second :: Integer -> Integer -> Integer  
second x y = y
```

second (div 1 0) 2

→ {- definición de second -}

2

Orden Normal

Se reduce el redex más externo y más a la izquierda.

La función es aplicada antes de evaluar sus argumentos

Pros

Sólo se evalúan las expresiones necesarias.

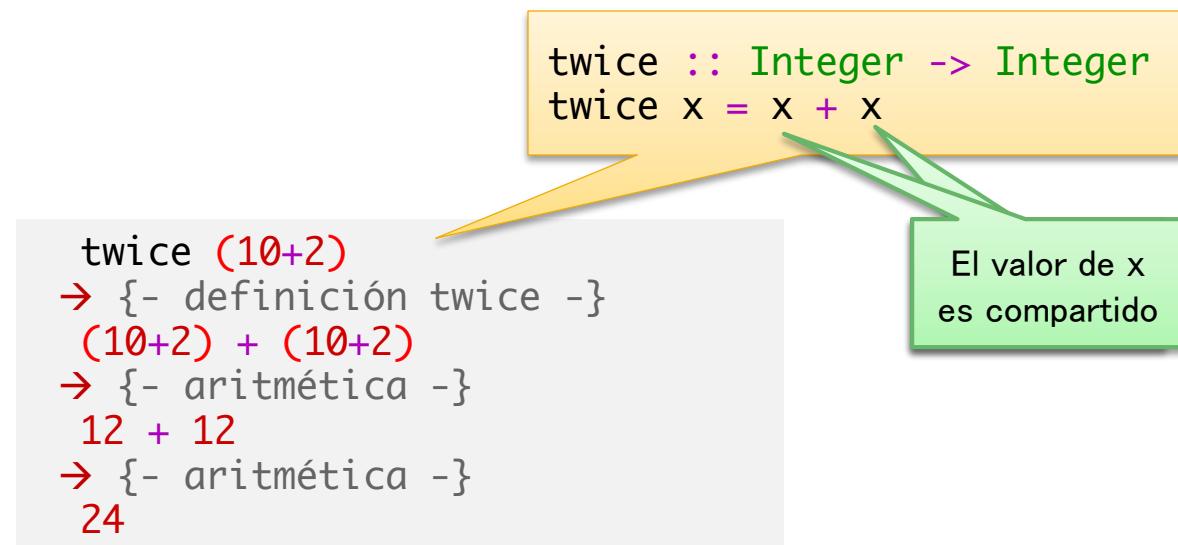
Siempre conduce a la forma normal si ésta existe (**normalizante**)

Contras

Puede repetirse la evaluación de un argumento compuesto

Evaluación de expresiones (VII)

- Haskell usa **evaluación perezosa (lazy)** :
 - Combina las ventajas del orden normal y del aplicativo
 - Es similar al orden normal pero evita la reevaluación:
 - Los argumentos son evaluados una sola vez y su valor común es compartido (**sharing**)
 - Esta optimización es posible en lenguajes puros



Tuplas

- Son estructuras de datos heterogéneas:
 - Es una colección de valores (**componentes**)
 - Cada componente puede tener un tipo distinto
- Sintaxis:
 - Los valores se separan por comas y se agrupan entre paréntesis
 - Los tipos de cada componente se separan por comas y se agrupan entre paréntesis

`(True, 2) :: (Bool, Int)`

`(10, 2, 7) :: (Int, Int, Int)`

Tuplas (II)

```
succPred :: Int -> (Int, Int)
succPred x = (x+1, x-1)
```

```
succPred 10
→ {- definición succPred -}
(10+1, 10-1)
→ {- aritmética -}
(11, 9)
```

Tuplas (III) y polimorfismo

- Funciones predefinidas que extraen las componentes de una tupla:

Prelude

Debe leerse como:
 $\forall a, b . (a, b) \rightarrow a$

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

a y b son **variables de tipo**, (en minúsculas) fst toma una 2-tupla con componentes de tipos arbitrarios a y b. Se dice que fst es **polimórfica**.

Patrón para una
2-tupla

Prelude

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

Indica una
función
predefinida

`fst (1, True)`
 $\rightarrow \{-\text{ definición fst }-\}$
1

`snd (1, True)`
 $\rightarrow \{-\text{ definición snd }-\}$
True

Sobrecarga. Tipos numéricos

- Una **clase de tipos** es un conjunto de tipos que comparten alguna operación
http://en.wikipedia.org/wiki>Type_class
- Num es la clase predefinida para los tipos numéricos (enteros, flotantes, racionales, ...)
- Algunos operadores y funciones solamente tienen significado para tipos numéricos:

debe leerse como:
 $\forall a \in \text{Num} . a \rightarrow a \rightarrow a$

Prelude

(+), (-), (*) :: $(\text{Num } a) \Rightarrow a \rightarrow a \rightarrow a$

tipo sobrecargado
a debe ser una
instancia de Num, es
decir, debe ser un tipo
numérico

Los dos operandos y el
resultado tienen que ser del
mismo tipo

Sobrecarga. Tipos numéricos (II)

```
Prelude> 1 + 2
```

suma sobre
Integer

3

```
Prelude> 3.5 * 2.5
```

producto sobre
Double

8.75

```
Prelude> 3.5 * 2
```

2 está sobrecargado

7.0

```
Prelude> 'a' + 'b'
```

Char no es de la clase Num
Type error



```
<interactive>:1:0:
```

No instance for (Num Char)

arising from a use of `+' at <interactive>:1:0-8

Possible fix: add an instance declaration for (Num Char)

In the expression: 'a' + 'b'

In the definition of `it': it = 'a' + 'b'

Aritmética entera

- **Integral** es la clase que agrupa los tipos que representan números enteros
- Define funciones específicas para la división entera:

Tipo sobrecargado: a
debe ser de la clase
Integral

debe leerse como:

$\forall a \in \text{Integral} . a \rightarrow a \rightarrow a$

Prelude

div, mod :: (Integral a) => a -> a -> a

cociente y resto de la
división entera

```
Prelude> div 10 3
3
Prelude> mod 10 3
1
Prelude> div 10.5 2.5
<interactive>:1:9:
```

Los números flotantes no
son Integrales.

Type error

Ambiguous type variable `t' in the constraints:
'Fractional t'
arising from the literal '2.5' at <interactive>:1:9-11
'Integral t' arising from a use of `div' at <interactive>:1:0-11
Probable fix: add a type signature that fixes these type variable(s)

Aritmética fraccionaria

- **Fractional** es la clase que agrupa los tipos de números fraccionarios: flotantes y racionales (cocientes de enteros , ...)
- La función esencial proporcionada es la división fraccionaria:

El tipo a debe ser de
la clase Fractional

Esto debe leerse como:
 $\forall a \in \text{Fractional} . a \rightarrow a \rightarrow a$

(/) :: (**Fractional** a) => a → a → a

División
fraccionaria

Prelude

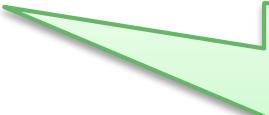
Prelude> 26.25 / 7.5
3.5

Prelude> 10 / 3
3.3333333333333335

10 y 3 están
sobrecargados.
Aquí actúan
como flotantes

Definición de funciones sobrecargadas

```
twice' :: (Num a) => a -> a
twice' x = x + x
```



Esta función usa el operador
sobrecargado **+**

Solo puede ser aplicada a tipos
numéricos; la función hereda el
contexto **Num a**

```
Main> twice' 10
20
```

```
Main> twice' 3.25
6.5
```

```
Main> twice' True
```

```
<interactive>:1:0:
No instance for (Num Bool)
```

arising from a use of `twice'' at <interactive>:1:0-4

Possible fix: add an instance declaration for (Num Bool)

In the expression: twice' True

In the definition of `it': it = twice' True



Tipos con orden

- **Ord** es la clase de los tipos para los cuales existe una relación de orden total; entre éstos están Integer, Double, Char, Rational, Bool, ...
- Ciertas operaciones ($<$, \max , ...) solo tienen sentido para las instancias de **Ord**:

Prelude

`(<), (<=), (>), (>=) :: (Ord a) => a -> a -> Bool`

Debe leerse como:
 $\forall a \in \text{Ord} . a \rightarrow a \rightarrow \text{Bool}$

`a` debe ser un tipo
instancia de **Ord**

Solo podemos comparar valores del
mismo tipo, y el resultado siempre
será un booleano

Prelude

`compare :: (Ord a) => a -> a -> Ordering`

`data Ordering = LT | EQ | GT`

máximo y mínimo de dos
valores del mismo tipo,
instancia de **Ord**

Tipo enumerado
predefinido: *less than*,
equal y *greater than*

Prelude

`max, min :: (Ord a) => a -> a -> a`

Tipos con igualdad

- **Eq** es la clase de los tipos para los cuales existe una relación de igualdad; entre éstos están **Integer**, **Double**, **Char**, **Rational**, **Bool**, ...
- La clase define dos operadores:

Debe leerse como:
 $\forall a \in Eq . a \rightarrow a \rightarrow Bool$

(==), (/=) :: (Eq a) => a → a → Bool

a debe ser un tipo
instancia de Eq

Solo podemos comparar
valores del mismo tipo, y el
resultado siempre será un
booleano

Ejemplo de diálogo

Prelude> $1 < 2$

1 es menor que 2

True

Prelude> $1 == 2$

1 no es igual a 2

False

Prelude> $1 /= 2$

1 es distinto de 2

True

Prelude> compare 1 2

1 es *Less Than* 2

LT

Prelude> compare 10 2

10 es *Greater Than* 2

GT

Prelude> compare 1 1

1 es *EQual* a 1

EQ

Prelude> 'a' > 'z'

False

Prelude> max 3.5 7.8

El mayor de 3.5 y 7.8 es 7.8

7.8

Prelude> min 3.5 2.0

El menor de 3.5 y 2.0 es 2.0

2.0

Clases y sobrecarga. Resumen

Eq `(==)`, `(/=)`

Ord
`(<)`, `(<=)`, `(>)`, `(>=)`
min, max, compare

Num `(+)`, `(-)`, `(*)`

Integral
div mod

Fractional `(/)`

Ecuaciones con guardas

- A menudo es más sencillo describir una función *por partes* (distintos casos):

sign $5 \rightarrow 1$
sign $0 \rightarrow 0$

sign $(-2) \rightarrow -1$



sign :: (Eq a, Ord a, Num a) $\Rightarrow a \rightarrow a$
sign x | $x > 0$ = 1
| $x < 0$ = -1
| $x == 0$ = 0

Atención al sangrado. Las barras verticales en la misma columna



Guardas booleanas

Resultado es 1, si el argumento es positivo
Resultado es -1, si el argumento es negativo

Resultado es 0, si el argumento es nulo

[http://en.wikipedia.org/wiki/Guard_\(computing\)](http://en.wikipedia.org/wiki/Guard_(computing))

Ecuaciones con guardas (II)

- La definición anterior:

```
sign :: (Eq a, Ord a, Num a) => a -> a
sign x | x > 0 = 1
      | x < 0 = -1
      | x == 0 = 0
```

- puede escribirse de forma más corta (**estilo preferente**):

```
sign :: (Ord a, Num a) => a -> a
sign x | x > 0 = 1
      | x < 0 = -1
      | otherwise = 0
```

otherwise: esta condición siempre es cierta. Será seleccionada si fallan las previas

Prelude **otherwise :: Bool**
otherwise = True

Ecuaciones con guardas y expresiones condicionales

- Las expresiones condicionales proporcionan una alternativa a las formas guardadas:

Prelude

```
abs :: (Ord a, Num a) => a -> a  
abs x = if x >= 0 then x else -x
```



-- versión con guardas

```
abs :: (Ord a, Num a) => a -> a  
abs x | x >= 0      = x  
      | otherwise    = -x
```

Funciones parciales

- Hay funciones definidas solo parcialmente:
- **Main> reciprocal 0**
*** Exception: reciprocal is undefined
- Esto se consigue vía la función error:

```
reciprocal :: (Eq a, Fractional a) => a -> a
reciprocal x | x == 0      = error "reciprocal is undefined"
              | otherwise   = 1 / x
```

el valor de tipo **String**
describe la excepción

- Error es una función polimórfica:

Prelude error :: **String** -> a

Definiciones locales where

- La palabra reservada **where** permite definir funciones o variables locales, y debe aparecer al final de la definición

```
circArea :: Double -> Double  
circArea r = pi*r^2
```

```
rectArea :: Double -> Double -> Double  
rectArea b h = b*h
```

```
circLength :: Double -> Double  
circLength r = 2*pi*r
```

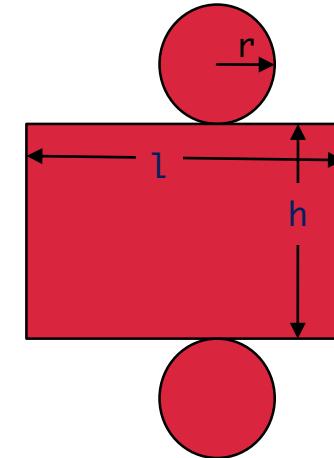
```
cylinderArea :: Double -> Double -> Double  
cylinderArea r h = 2*circ + rect
```

where

```
circ = circArea r  
l = circLength r  
rect = rectArea l h
```

where: debe sangrarse con respecto a la función

Las **definiciones locales** deben sangrarse al mismo nivel



Declaraciones de operadores

infix = no asociativo

infixl = asociativo a izda

infixr = asociativo a dcha

Nivel de precedencia

infix 4 $\sim=$ -- Comprueba si dos reales son
-- aproximadamente iguales

$(\sim=)$:: Double -> Double -> Bool

x $\sim=$ y = abs (x-y) < epsilon

where epsilon = 1/1000

El identificador del operador debe aparecer entre paréntesis en la declaración de tipo

```
Main> (1/3)  $\sim=$  0.33
False
Main> (1/3)  $\sim=$  0.333
True
```

Pruebas con QuickCheck

- Es una buena práctica probar los programas
- Las pruebas ayudan a encontrar errores 
- `Test.QuickCheck` es una biblioteca de Haskell que ayuda a probar los programas
 - Definimos propiedades que nuestros programas deben cumplir
 - `Test.QuickCheck` genera automáticamente casos de prueba y verifica las propiedades para esos casos



Pruebas con QuickCheck (II)

■ Propiedades en QuickCheck:

antecedente ==> consecuente

- *antecedente* y *consecuente* deben ser expresiones booleanas
- Debe leerse como una implicación lógica: si el *antecedente* es cierto entonces el *consecuente* también debe serlo
- QuickCheck solo realiza las verificaciones para los casos en los que el *antecedente* es cierto. Las pruebas generadas cuyo *antecedente* es falso son descartadas
- Se puede usar **True** si no hay *antecedente*:
True ==> consecuente

Pruebas con QuickCheck (III)

■ Algunas propiedades:

```
square :: (Num a) => a -> a
square x = x * x
```

$$\forall x. \forall y . (x + y)^2 = x^2 + y^2 + 2xy$$

p1 $x = \text{True} \implies \text{square}(x+y) == \text{square } x + \text{square } y + 2*x*y$

$$\forall x. \forall y . |x + y| = |x| + |y|$$

p2 $x = \text{True} \implies \text{abs}(x+y) == \text{abs } x + \text{abs } y$

$$\forall x \geq 0. \forall y \geq 0 . |x + y| = |x| + |y|$$

p3 $x = x \geq 0 \ \&& \ y \geq 0 \implies \text{abs}(x+y) == \text{abs } x + \text{abs } y$

Hacemos que QuickCheck pruebe nuestra propiedad con enteros aleatorios

```
Main> quickCheck (p1 :: Integer -> Integer -> Property)
+++ OK, passed 100 tests.
```

```
Main> quickCheck (p2 :: Integer -> Integer -> Property)
*** Failed! Falsifiable (after 2 tests):
```

```
1  
-1
```

```
Main> quickCheck (p3 :: Integer -> Integer -> Property)
+++ OK, passed 100 tests.
```

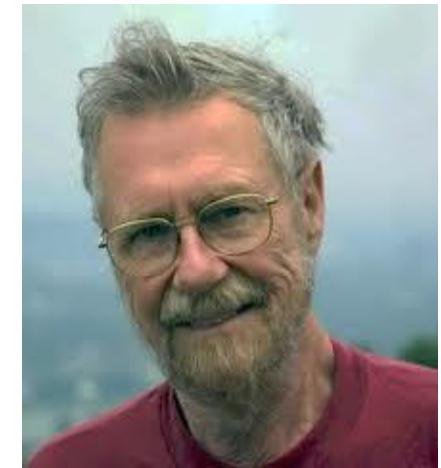
Fallo en la prueba:

Contraejemplo cuando
 x es 1 e y es -1

Pruebas con QuickCheck (y IV)

- **Edsger W. Dijkstra** (1930-2002, Premio Turing 1972)

“¡Las pruebas de un programa
pueden usarse para mostrar la
existencia de errores, pero nunca
para demostrar su ausencia!”



http://en.wikipedia.org/wiki/Edsger_W._Dijkstra

- Para establecer la corrección de un algoritmo se debe realizar una demostración
- En bastantes ocasiones es difícil y lleva mucho tiempo
- Realizar pruebas es mejor que no hacer nada 😊

Material Complementario

Para profundizar

Definiciones locales let in

- Una alternativa para introducir definiciones locales es vía la construcción **let ... in**

```
circArea :: Double -> Double  
circArea r = pi*r^2
```

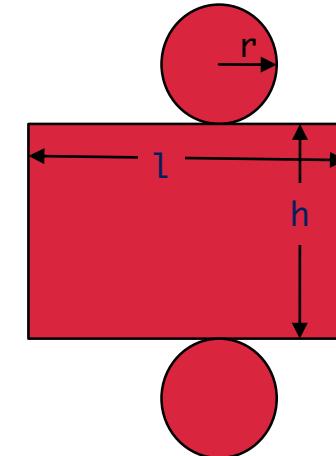
```
rectArea :: Double -> Double -> Double  
rectArea b h = b*h
```

```
circLength :: Double -> Double  
circLength r = 2*pi*r
```

```
cylinderArea :: Double -> Double -> Double  
cylinderArea r h =
```

```
let  
  circ = circArea r  
  l    = circLength r  
  rect = rectArea l h
```

```
in  
  2*circ + rect
```



Las **definiciones locales**
deben sangrarse al
mismo nivel

Resultado de la
expresión **let ...in**