Tema 3: Paradigma Funcional

Lenguajes, Tecnologías y Paradigmas de Programación

Escuela Técnica Superior de Ingeniería **Informática**

etsinf

Índice

Introducción a la Programación Funcional

PARTE I: Tipos en Programación Funcional

1. Tipos funcionales. Tipos algebraicos.
2. Tipos predefinidos.
3. Polimorfismo: genericidad, sobrecarga y coerción. Herencia en Haskell.

PARTE II: Modelos de computación funcional

4. Modelo operacional.

PARTE III: Características avanzadas

5. Funciones anónimas y composición de funciones.
6. Iteradores y compresores (foldl, foldr).

Objetivos

- Identificar los **fundamentos del paradigma funcional**: transparencia referencial y ausencia de efectos laterales y variables globales, funciones como ciudadanos de primera clase.
- Comprender los **tipos algebraicos y funcionales** de los lenguajes funcionales modernos.
- Conocer la **relación entre polimorfismo y herencia,** y su uso en lenguajes funcionales y orientados a objetos.
- Conocer y resolver problemas utilizando **funciones parciales y que no terminan**.
- Comprender y saber aplicar la **currificación,** la **aplicación parcial** y el **orden superior**.
- Comprender y saber aplicar el **modelo de computación funcional** basado en reducción y estrategias de evaluación.
- Saber aplicar **esquemas de iteración y compresión** en la resolución de problemas.
- Entender el esquema "**mapreduce**" y su conexión con el paralelismo. Comprender cómo se utiliza para el procesamiento de información en la web.

Introducción

Soluciones a la crisis del Software

- nuevos desarrollos de la Ingeniería del Software que permitan solucionar el problema del análisis y diseño de grandes proyectos informáticos
- proporcionar sistemas de prueba y verificación de programas
- desarrollar técnicas de síntesis de programas para poder obtener código ejecutable a partir de especificaciones formales
- diseñar nuevas arquitecturas de computadoras (técnicas de procesamiento paralelo)
- **proponer un modelo de computación diferente al modelo imperativo tradicional**

Impacto de la programación funcional

- Por un lado, los lenguajes funcionales se usan cada día en más ámbitos:
  - **Haskell** (funcional puro, en muchos campos, ver https://wiki.haskell.org/Haskell_in_industry)
  - **Scala** (funcional & OO, componentes de Twitter)
  - **Erlang** (funcional & concurrente, fundamental en aplicaciones como Whatsapp o Facebook, ver http://www.wired.com/2015/09/whatsapp-serves-900-million-users-50-engineers/)

Impacto de la programación funcional

- Por otro lado, cada vez más lenguajes incorporan características de la programación funcional:
  - **Python**: orden superior, map, reduce, etc.
  - **JavaScript**: orden superior, abstracciones lambda, closures, map, etc.
  - **Java**: orden superior, abstracciones lambda
  - **Ruby**: orden superior, abstracciones lambda, aplicaciones parciales
  - **PHP**: orden superior, abstracciones lambda, etc.

Algunas características distintivas

- Ausencia de efectos laterales
- Funciones como ciudadanos de primera clase
- Tipos y estructuras de datos de usuario
- Aplicación parcial
- Estrategias de evaluación

Ausencia efectos laterales - Funciones

- Ausencia de efectos laterales y funciones como ciudadanos de primera clase
  - Ausencia de efectos laterales

    El resultado devuelto por una función *sólo* depende de sus argumentos de entrada (transparencia referencial)
  - Ciudadanos de primera clase

    Una función puede ser pasada como argumento (orden superior) o devuelta como resultado de la computación (por ejemplo, mediante aplicación parcial).

```
: map sqr [1,2,3]
[1,4,9]
```

```
: map (inc 1) [1,2,3]
[2,3,4]
```

donde:

```
inc:: Int -> (Int -> Int)
inc x = (x +)
```

Aplicación parcial de funciones

- A cada función

$$f : D_1 \times D_2 \times \ldots \times D_k \rightarrow E$$

le corresponde una función

$$f' : D_1 \rightarrow (D_2 \rightarrow (\ldots \rightarrow (D_k \rightarrow E) \ldots))$$

que a cada valor de D_1 le asocia una función de (k-1) argumentos (y así sucesivamente)

- Esto se llama **currificación** y hace posible la **aplicación parcial** de una función.

Aplicación parcial de funciones

- La aplicación parcial de una función consiste en invocarla con un número de argumentos menor que el número de parámetros de su definición.
- Ejemplo: operadores aritméticos

```
(+)  :: Int -> Int -> Int
: (2 +) 5
7
```

- Ejemplo: otras funciones definidas por aplicación parcial

```
sumar_2 :: (Int -> Int)
sumar_2  = (2+)
: sumar_2 5
7
```

Estrategias de evaluación

- Estrategias de evaluación

```
tres     :: Int → Int
tres x   = 3

infinito :: Int
infinito = infinito +1
```

```
    tres infinito
=   {definición de infinito}
    tres (infinito +1)
=   {definición de infinito}
    tres ((infinito +1)+1)
=   {y así sucesivamente}
    . . .
```

evaluación impaciente: evalúa todos los argumentos de una función antes de conocer si serán necesarios.

```
    tres infinito
=   {definición de tres}
    3
```

evaluación perezosa: no evalúa los argumentos de una función si no es necesario. Garantiza la terminación siempre que sea posible.

Índice

Tipos funcionales

- El constructor de tipos -> permite crear tipos funcionales a partir de otros tipos.

  Ejemplo: `type MyType = (Int -> Int)`

  `fib :: MyType`

- En general a_1 -> a_2 -> … -> a_n es un tipo funcional cuyos valores son todas las funciones que tienen ese tipo.
  - por ejemplo, la función not es un valor del tipo Bool -> Bool
  - la función (2+) pertenece al tipo Int -> Int
  - la función map pertenece al tipo (a -> b) -> [a] -> [b]

¿y la función map (2+) a qué tipo pertenece?

- El operador -> es **asociativo por la derecha**

  a -> b -> c equivale a a -> (b -> c)

  y difiere de (a -> b) -> c

- El operador de aplicación funcional es asociativo por la izquierda

  f a b equivale a (f a) b y difiere de f(a b)

Ejemplo

: not not False

Tipos algebraicos (ejemplos)

```
data Estado = Casado | Soltero

estado :: Estado -> String
estado Casado = "Ha pasado por el altar"
estado Soltero = "Libre como un pájaro"

: estado Soltero
"Libre como un pájaro”
```

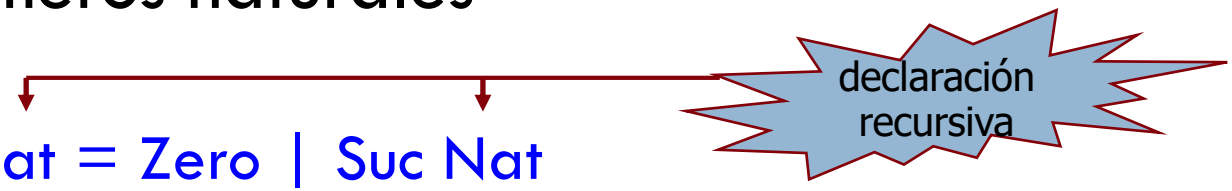
```
data Estado = Casado Bool | Soltero Int

estado :: Estado -> String
estado (Casado x) = if x then "Es feliz" else "Es infeliz"
estado (Soltero x) = "Le faltan "++(show x)++" años para casarse"

:estado (Casado True)
”Es feliz"
```

Tipos algebraicos (ejemplos)

```
type Nombre = String
type Cargo = String
type Edad = Int
type Curso = Int

data Persona = Alumno Edad Nombre Curso |
               Profesor Nombre Cargo |
               Rector Nombre

nombrePersona :: Persona -> Nombre
nombrePersona  (Alumno e n c) = n
nombrePersona  (Profesor n c) = n
nombrePersona  (Rector n) = n

: nombrePersona (Profesor "Alberto" "Ayudante")
"Alberto"
```

Tipos algebraicos (ejemplos)

- Los números naturales




```
data Nat = Zero | Suc Nat
```

  - Los valores: Zero, Suc(Suc Zero), Suc(Suc(Suc(Suc(Suc Zero)))),…

representan los números 0, 2, 5, …

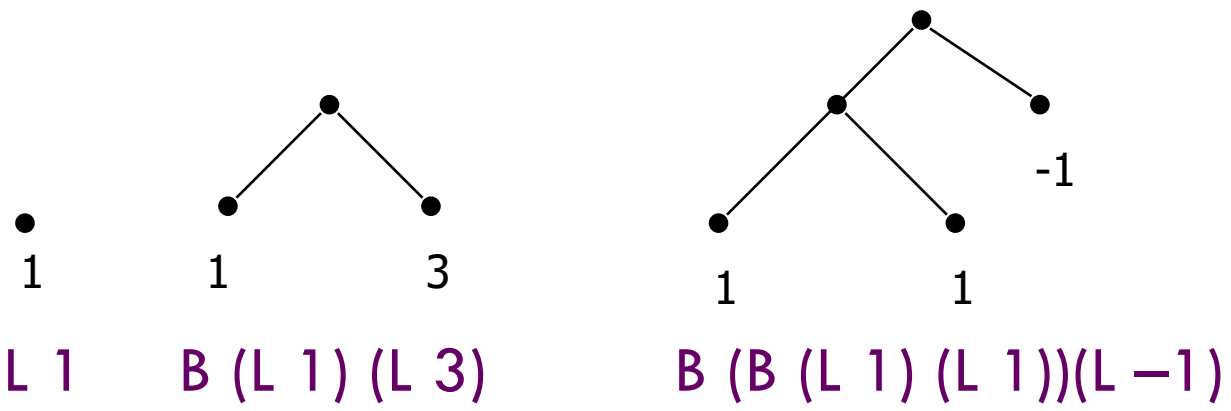
  - El **patrón** Suc(n) representa los números naturales ***positivos***
  - Ejemplo de operaciones sobre los naturales: suma y producto

```
(+$) :: Nat -> Nat -> Nat
x +$ Zero = x
x +$ (Suc y) = Suc (x +$ y)
(*$) :: Nat -> Nat -> Nat
x *$ Zero = Zero
x *$ (Suc y) = x +$  (x *$  y)
```

Tipos algebraicos (ejemplos)

- **Ejemplo:** El tipo de los árboles binarios de elementos de tipo entero se puede definir como

```
data BinTreeInt = L Int | B (BinTreeInt) (BinTreeInt)
```



Tipos algebraicos

- enumeraciones

simples

```
data Dia = Dom | Lun | Mar | Mie | Jue | Vie | Sab
```

- tipos cuyos valores dependen de los valores de otros tipos

estruc-turados

```
data Either = Left Bool | Right Char
```

constructoras de dato

en general

paramé-tricos

```
data Either a b = Left a | Right b
```

variables de tipo

uso de expresiones con Left y Right como patrones

```
either :: (a -> c) -> (b -> c) -> Either a b  -> c
either f g (Left x) = f x
either f g (Right y) = g y
```

predefinido

$$\textbf{data T } a_1 \ldots a_n = C_0\ t_{01} \ldots t_{0k_0} \mid \ldots \mid C_m\ t_{m1} \ldots t_{mk_m}$$

constructor de tipo

Tipos algebraicos: valores/patrones

- Los valores de un tipo algebraico son expresiones constituidas por símbolos constructores de datos.
- Se obtienen utilizando la definición del tipo como una gramática, donde:
  - los constructores de datos => símbolos terminales
  - los constructores de tipo => símbolos no terminales
  - P.e.: Zero, Suc Zero, Suc (Suc Zero),…
- Los patrones son expresiones constituidas por símbolos constructores de datos y variables
  - Representan conjuntos de valores. Por ejemplo, (Suc n) puede representar el conjunto de números naturales positivos

Tipos algebraicos: ajuste de patrones

- Una expresión e **se ajusta a** un patrón *p* (*pattern matching*) si e puede verse como una **concreción** de *p* (dando ciertos valores a las **variables** de *p*)
- El ajuste de patrones es un mecanismo habitual para definir funciones en programas funcionales
- Podemos definir una función describiendo cómo actúa ante **conjuntos de valores** descritos mediante un **patrón**

- Algunas funciones definidas por *ajuste de patrones*:
  - Or exclusivo:

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

  - if _ then _ else

```
cond   :: Bool -> a -> a -> a
cond True x y = x
cond False x y = y
```

Índice

Tipos predefinidos (Char)

- se denotan encerrándolos entre comillas simples

  'a', 'b', '0', '\n', '\r',...

- Funciones primitivas para procesar caracteres:

  ord :: Char -> Int *convierte un carácter c a un entero ord c*

  chr :: Int -> Char *convierte un entero en el carácter que este representa*

  operadores relacionales

  ```
  : ord 'b'
  98
  : chr 98
  'b'
  : 'A' < 'a'
  True
  : 'b' == chr 98
  True
  ```

- Otras funciones:
  - isAlpha, isAlphaNum, isDigit, isLower, isUpper :: Char -> Bool
  - toLower, toUpper :: Char -> Char
  - putChar :: Char -> IO ()

Tipos predefinidos (tupla)

- Una tupla es un tipo de datos compuesto cuyas componentes pueden tener distinto tipo.

  (Int,Char)

  (Char,(Int,Char))

  (Char,Int,Char)

  . . .

- Funciones para tuplas de 2 componentes (pares)
  - fst :: (a,b) -> a

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

    snd (x,y) = y

Ejercicio

- Definid una función que dados dos números los devuelva ordenados de menor a mayor.

Ejercicios

1. Define una función siglet :: Char -> Char que transforme cada letra del alfabeto en la letra siguiente, dejando invariantes los restantes caracteres. Suponemos que debe cumplirse siglet 'Z' = 'A' y siglet 'z' = 'a'.

2. Supongamos que se representa una fecha como una tupla (d,m,a) de números enteros, correspondientes al día, mes y año respectivamente. Define una función que calcule la edad de un individuo en años, dadas la fecha de su nacimiento y la fecha actual.

3. Sean sigma y pi las funciones especificadas como sigue:

   sigma f a b = $\Sigma_{a \le i \le b}$ f i

   pi f a b = $\Pi_{a \le i \le b}$ f i

   construye definiciones recursivas ejecutables para las dos funciones, incluyendo declaraciones de tipo.

Tipos predefinidos (String)

- Definición: **type** String = [Char]
  - se representan usando comillas dobles
  - se pueden hacer comparaciones siguiendo el orden lexicográfico

    : “Juan” < “Juana”

    True

    : “Palo” < “palo”

    True
  - Para transformar datos de ciertos tipos al tipo cadena

    show :: a-> String

    : show 6

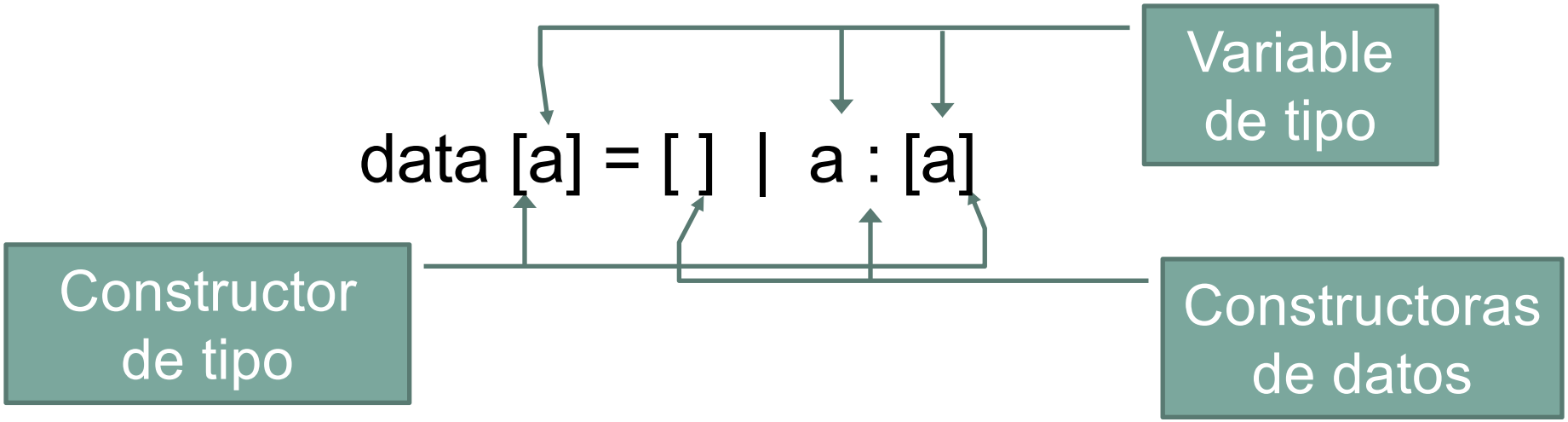
    “6”

Tipos predefinidos (númericos)

- Tipos numéricos: Int, Float
  - **Int**: enteros de rango acotado
  - **Float**: reales en coma flotante de precisión simple (0.345, -23.12, 231.61e7, 46.7e-2,...)
- En Haskell existen otros tipos de datos numéricos
  - **Integer**: enteros de rango arbitrario
  - **Double**: reales en coma flotante de precisión doble
  - **Complex**: números complejos
  - **Rational**: números racionales (en la librería Ratio)

Tipos predefinidos (listas)

El tipo predefinido *lista* corresponde a un tipo algebraico polimórfico recursivo que podría expresarse así:



Notación

- Por su frecuente uso en programación funcional, se han desarrollado (y se admiten) distintas notaciones para expresar listas:

  1:2:3:[] (igual a 1:(2:(3:[])))

  [1,2,3]

  1:[2,3]

  corresponden a la misma lista

(:) es *asociativo por la derecha*

- La notación de **listas aritméticas** permite expresar secuencias de valores de tipos enumerados

[2..10]	es	[2,3,4,5,6,7,8,9,10]
[1..]	es	[1,2,3,4,...
[1,3..10]	es	[1,3,5,7,9]
['a'..'e']	es	"abcde"

Listas intensionales

La notación de **listas intensionales** se inspira en las expresiones utilizadas habitualmente en matemáticas para expresar conjuntos.

Ejemplo. El conjunto de los cuadrado de los enteros comprendidos entre el 1 y el 5

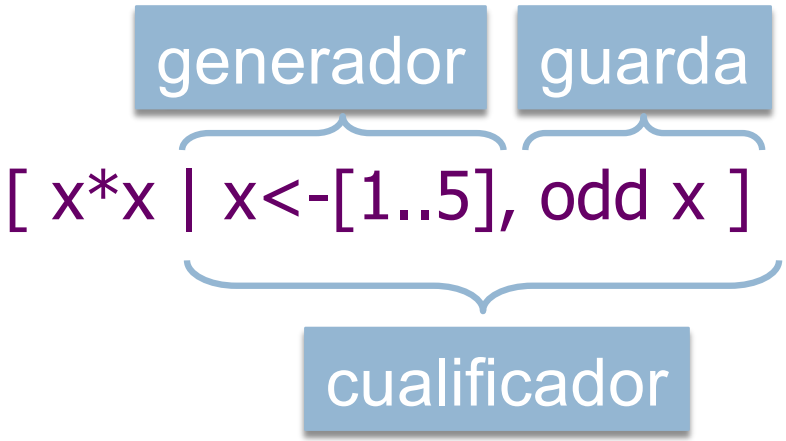
{x*x | 1<=x<=5 , impar x}

[x*x | x<-[1..5], odd x]

lista aritmética

Listas intensionales

La notación de **listas intensionales** se inspira en las expresiones utilizadas habitualmente en matemáticas para expresar conjuntos.

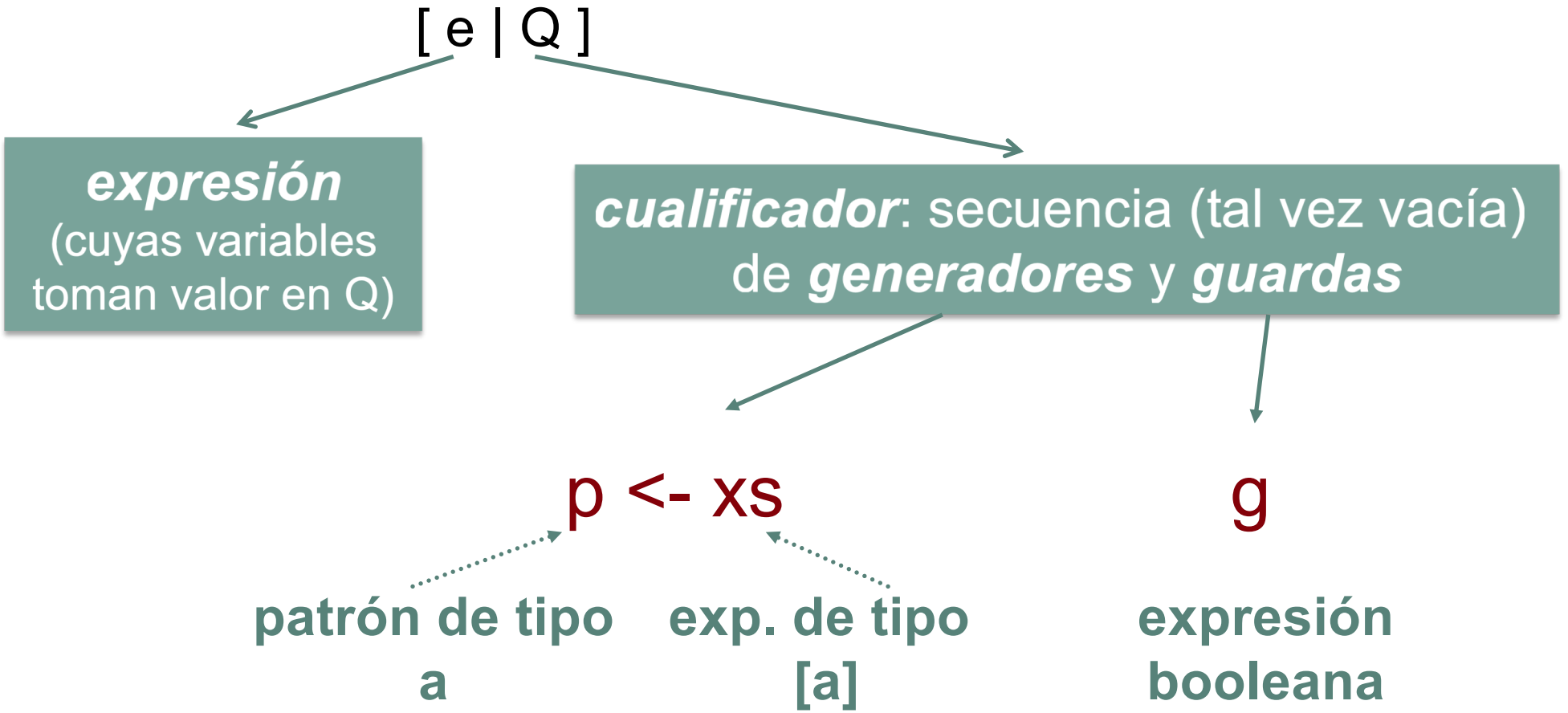


Significado:
Lista con todos los resultados de multiplicar x*x.
Donde x es un valor entero impar entre 1 y 5

Resultado:
[1*1, 3*3, 5*5] = [1, 9, 25]

Listas intensionales

Sintaxis de las listas intensionales :



Listas intensionales

- Semántica de las listas intensionales:

  [e | p_1<-xs_1, g_1,..., p_n<-xs_n, g_n]

  - los generadores actúan de izquierda a derecha, variando primero el de más a la derecha
  - las guardas se evalúan de izquierda a derecha
  - la lista devuelta consta de los valores obtenidos al evaluar e con las variables generadas, si se satisfacen todas las guardas

: [(x,y) | x<-[1..5], odd x, y<-[x..5], odd y]

[(1,1), (1,3), (1,5), (3,3), (3,5), (5,5)]

: [(x,y) | (x:y:xs) <- ["abcde", "f", "ghi"]]

[('a','b'),('g,'h')]

Ejercicio: Definid las funciones sigma y pi usando listas intensionales

Ejemplos

- map f xs = [f x | x <- xs]
- filter p xs = [x | x <- xs, p x]
- repetido y xs = length [()| x <- xs, y == x]
- divisores n = [i| i <- [1..n], n `mod` i == 0]
- pertenece y xs = not (null [()| x<-xs, y == x])

Ejercicios

1. Define una función

   elimDups :: [Int] -> [Int]

   que elimine de una lista los elementos duplicados adyacentes.

   Por ejemplo

   elimDups [1,2,2,3,3,3,1,1]=[1,2,3,1]

2. Define dos funciones

   any, all :: (a -> Bool) -> [a] -> Bool

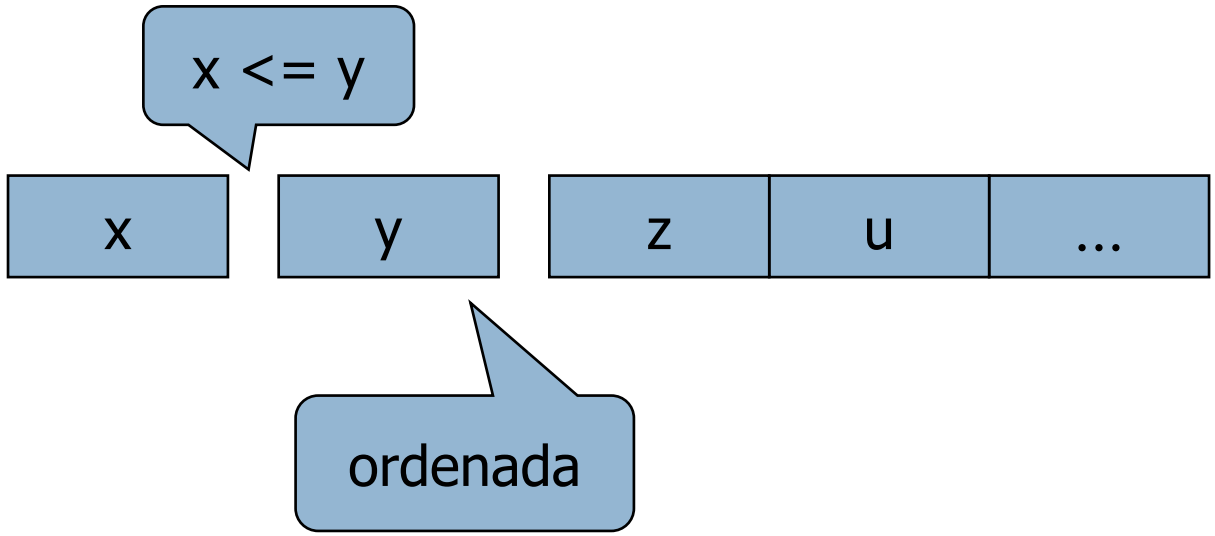
   que satisfagan las siguientes especificaciones

   any p xs $\Leftrightarrow \exists\ x \in xs : p\ x$

   all p xs $\Leftrightarrow \forall\ x \in xs : p\ x$

Ejercicios

- Define una función que compruebe si una lista de enteros está ordenada.



- Define una función que cree una lista de n copias de su parámetro x (Nota: usad listas intensionales).

Operaciones sobre listas

- Propiedades de una lista
  - Longitud de una lista: length :: [a] -> Int

    length [ ] = 0

    length (x:xs) = 1 + length xs

  - ¿Lista vacía?: null :: [a] -> Bool

    null [ ] = True

    null (x:xs) = False

Combinación de listas

- Concatenación: (++) :: [a] -> [a] -> [a]

```
[ ] ++ xs = xs
(x:xs) ++ ys = x:(xs ++ ys)
```

- Aplanamiento: concat :: [[a]] -> [a]

```
concat [ ] = [ ]
concat (xs:xss) = xs ++ concat xss
```

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

```
zip [ ] xs  = [ ]
zip (x:xs) [ ]      = [ ]
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

Ejemplo : zip [1, 2, 3] [”a”, ”b”, ”c”]

[(1,”a”), (2,”b”), (3,”c”)]

donde zip :: [Int] -> [String] -> [(Int, String)]

- Acceso a componentes de una lista
  - Cabeza de una lista: `head :: [a] -> a`

    `head (x:xs) = x`
  - Último de una lista: `last :: [a] -> a`

    `last [x] = x`

    `last (x:xs) = last xs`
  - Acceso posicional: `(!!) :: [a] -> Int -> a`

    `(x:xs) !! 0 = x`

    `(x:xs) !! (n) = xs !! (n-1)`
- Acceso a sublistas de una lista
  - Principio de una lista: `init :: [a] -> [a]` -- todos menos el último elemento
  - Resto de una lista: `tail :: [a] -> [a]` -- todos menos el primer elemento

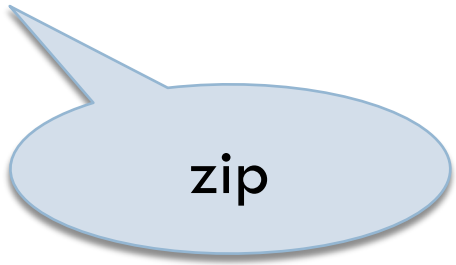
Ejercicio 1

- Definir una función posicion que indique la posición que ocupa un elemento en una lista.

```
position :: a -> [a] -> Int
: position "b"  ["a", "b", "c"]
2
```

Sugerencia: marcar cada elemento de la lista con su posición y localizar cuál es la del elemento buscado

["a", "b", "c"] -----> [("a",1), ("b",2), ("c",3)]



zip [”a”, ”b”, ”c”] [1, 2, 3]= [(”a”,1), (”b”,2), (”c”,3)]

- No sabemos cuántos elementos tiene la lista xs, pero sí sabemos que serán tantos como length xs

zip xs [1.. length xs]

Seleccionar la posición en la que aparece el elemento que estamos buscando

```
position x xs = [pos | (x',pos) <- zip xs [1..], x' == x]
```

```
position "b" ["a", "b", "c"] = [2]
```

Una lista cuando nosotros queremos un número

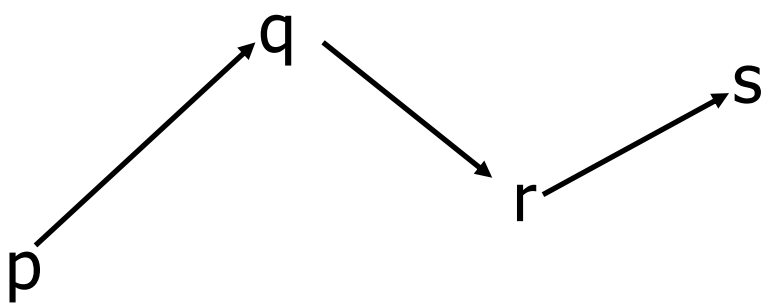
Solución:

```
position :: a -> [a] -> Int
position x xs = head [pos | (x',pos) <- zip xs [1..], x' == x]
```

selecciona el primer elemento de la lista

Ejercicio 2

- Definir una función que calcule la longitud de un camino.



- Representación:

```
type Point = (Float,Float)
type Path = [Point]
examplePath = [p,q,r,s]
```

```
pathLength = distance p q + distance q r + distance r s
```

- Dos funciones útiles:

```
init [p, q, r, s] = [p, q, r]
tail [p, q, r, s] = [q, r, s]
```

- Para combinar ambas listas: zip

```
zip …      = [(p,q), (q,r), (r,s)]
```

Solución:

```
pathLength :: Path -> Float

pathLength xs = sum' [distance p q | (p,q) <- zip (init xs) (tail xs)]


sum' :: [Float] -> Float
sum' [] = 0.0
sum' (x:xs) = x + sum' xs

distance :: Point -> Point -> Float
distance (p1,p2) (q1,q2) = sqrt (sqr(p1 -  q1) + sqr(p2 - q2))
```

Operaciones sobre listas

- Ordenación de listas
  - Por inserción en una lista ordenada

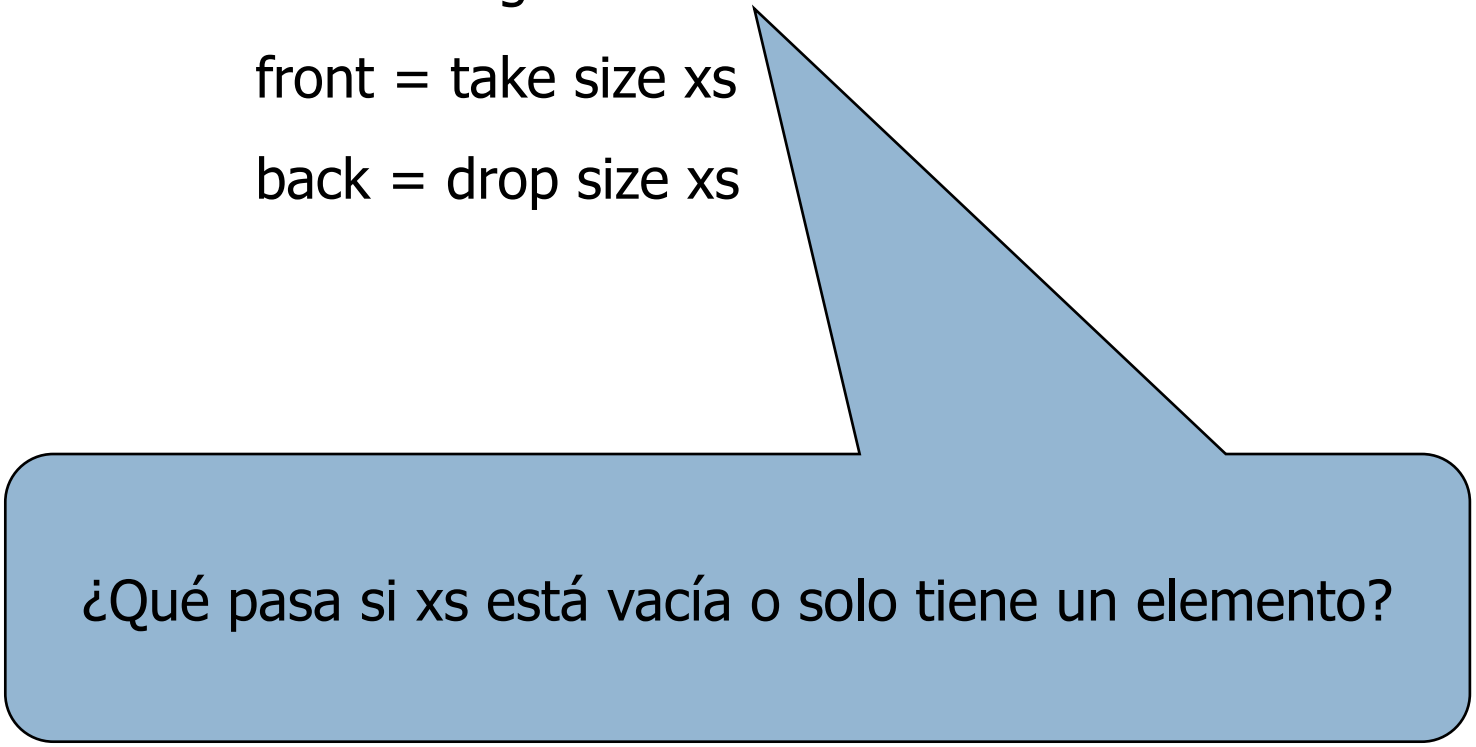
```
insertar x [ ] = [x]
insertar x (y:ys)
    | x<=y = (x:y:ys)                    -- [x]++(y:ys)
    | otherwise = y : (insertar x ys)

inorder [ ] = [ ]
inorder (x:xs) = insertar x (inorder xs)
```

- Una forma mejor de ordenar: **mergeSort**

1. dividir la lista en dos partes iguales
2. ordenar cada parte
3. poner juntas las dos mitades ordenadas

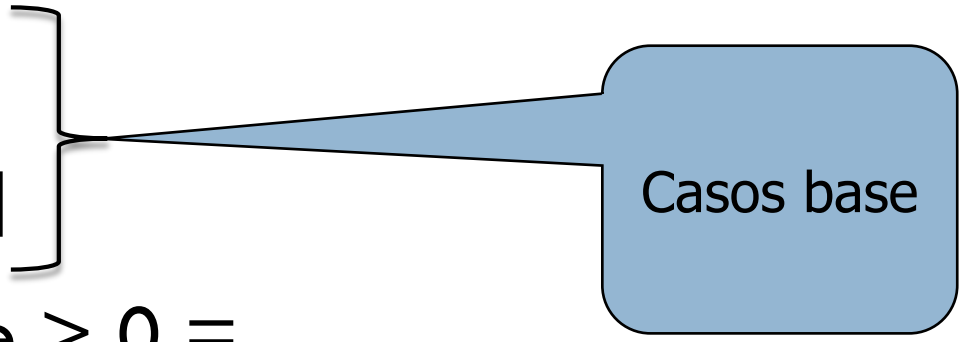
Ejemplo: Mergesort

```
mergeSort xs = merge (mergeSort front) (mergeSort back)
    where size = length xs `div` 2
          front = take size xs
          back = drop size xs
```

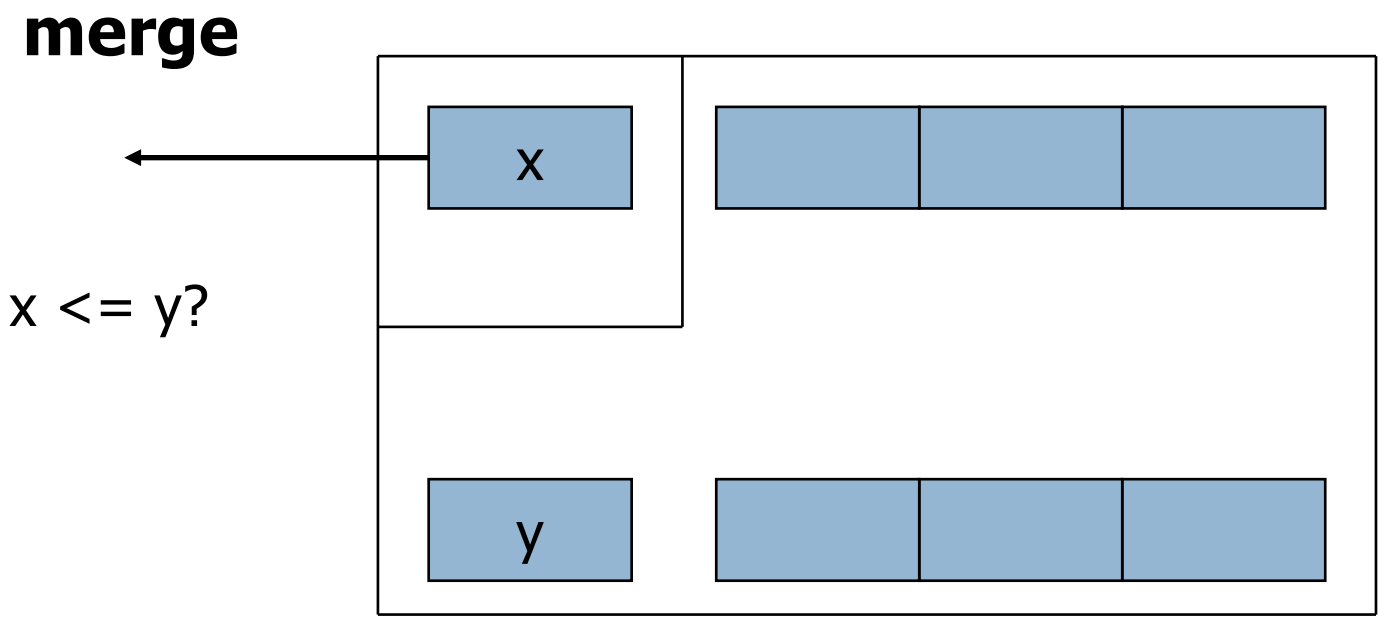


Ejemplo: Mergesort

```
mergeSort [] = []
mergeSort [x] = [x]
mergeSort xs | size > 0 =
    merge (mergeSort front) (mergeSort back)
    where     size = length xs `div` 2
              front = take size xs
              back = drop size xs
```



Ejemplo: Mergesort



merge [1, 3] [2, 4] ⟶ 1 : merge [3] [2, 4]

⟶ 1 : 2 : merge [3] [4]

⟶ 1 : 2 : 3 : merge [] [4]

⟶ 1 : 2 : 3 : [4] ⟶ [1,2,3,4]

Ejemplo: Mergesort

Solución:

```
merge :: [Int] -> [Int] -> [Int]

merge a@(x:xs)  b@(y:ys)
   | x <= y        = x : merge xs b
   | otherwise     = y : merge a  ys

merge [ ] ys       = ys

merge xs [ ]       = xs
```

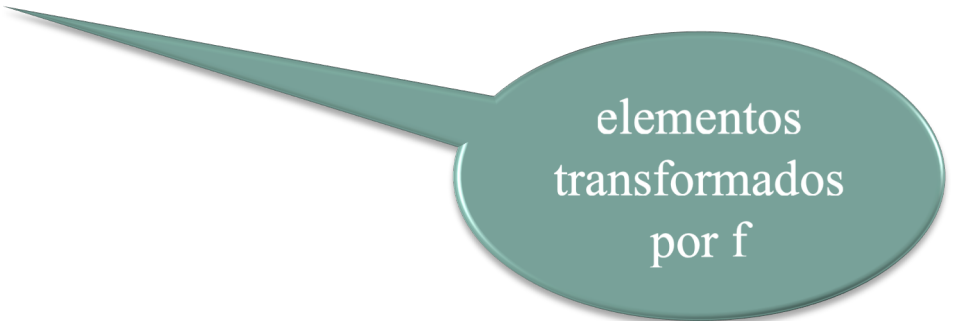
alias o seudónimo de un patrón

- Transformación de listas
  - Inversión: reverse :: [a] -> [a]

```
reverse [ ] = [ ]
reverse (x:xs) = reverse xs ++ [x]
```

  - Aplicando una función a sus elementos:

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



Índice

Coerción

- En Haskell la coerción es explícita y se hace mediante funciones que transforman unos tipos en otros.

Ejemplos:

- La conversión entre tipos numéricos tampoco es automática y hay funciones específicas para cada caso

```
import GHC.Float
:t float2Double                    :t double2Float
float2Double:: Float -> Double     double2Float:: Double-> Float
```

- la función show transforma a string cualquier tipo predefinido. Podemos usarlo para hacer la coerción de entero a string

show 3 =“3”

Genericidad

- Una función es genérica si su tipo es polimórfico (contiene variables de tipo).

  Ejemplo:

```
either :: (a -> c) -> (b -> c) -> Either a b -> c
either f g (Left x) = f x
either f g (Right y) = g y
```

Sobrecarga programable

- Sobrecarga en Haskell: implementada a través del concepto de **clase de tipo.**
  - El sistema de clases de Haskell permite utilizar la parametrización para definir funciones sobrecargadas, imponiendo a los tipos sobre los que se aplica la función la condición de pertenecer a una clase.

Ejemplo: `(+) :: Num a => a -> a -> a`

restringe la suma a los tipos que pertenecen a la clase Num.

  - La declaración de una **clase de tipo** (**class**) especifica cuáles son las operaciones que debe implementar cualquier tipo que pertenece o es instancia de dicha clase.

Ejemplo:
```
class Eq a where
    (==), (/=) :: a -> a -> Bool
    x/=y = not(x==y)
```

  - Una función sobrecargada recibe implícitamente en su tipo el contexto apropiado.

Ejemplo:
```
:t (==)
== :: Eq a => a->a->Bool
```

Sobrecarga programable

- El mecanismo **instance** permite introducir un tipo de datos en una clase indicando cómo implementa dicho tipo las operaciones de la clase.

Ejemplo:

```
data Nat = Zero | Suc Nat

instance Eq Nat where
  Zero == Zero = True
  Suc x == Suc y = x == y
  _ == _ = False
```

- Un tipo se puede declarar como instancia de varias clases usando la cláusula **deriving** en la definición del tipo.

Ejemplo: **data** Bool= False| True **deriving** (Eq, Ord, Enum)

donde las operaciones se basan en la propia definición del tipo (por ejemplo, False < True).

Algunas clases predefinidas

- Eq((==), (/=))

contiene todos los tipos predefinidos excepto IO, (->)

- Ord((<), (<=), (>=), (>), max, min)

contiene todos los tipos predefinidos excepto IO, IOError, (->)

- Num((+), (-), (*), negate, abs, signum, …)

contiene todos los tipos numéricos (Int, Integer, Float, Double, Ratio)

- Show(show,…)

contiene todos los tipos predefinidos excepto IO, (->)

Extensiones de clases

- las clases pueden extenderse mediante el propio mecanismo **class**.

Ejemplo: la clase Ord es una subclase de Eq que da una implementación por defecto de <=, >=, >

```
class  (Eq a) => Ord a  where

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

  x <= y       =  (x<y) || (x==y)
  x >= y       =  (x>y) || (x==y)
  x >  y       =  not (x <= y)
```

Extensiones de clases

- Las instancias de las clases extendidas pueden declararse mediante el mecanismo **instance**.

Ejemplo:

```
instance (Eq Nat) => Ord Nat where
  Zero < Suc x = True
  Suc x < Suc y = x < y
  _ < _ = False
```

Nota: el contexto (Eq Nat)=> no es necesario ponerlo ya que Ord es una extensión de Eq, pero es necesario que hayamos declarado Nat como instancia de la clase Eq antes de declarar que Nat es instancia de la clase Ord.

Instancias y derivadas

- Cuando los tipos son genéricos, es posible que necesitemos imponer contextos (pertenencia a clases) a sus argumentos.
- Ejemplo:

```
data Figure = Circle Float | Rect Float Float deriving (Eq, Ord, Show)
```

Float debe ser instancia de Eq, Ord, Show (lo es de forma predefinida)

```
data Tree a = Void | Branch a (Tree a) (Tree a)

instance (Eq a) => Eq (Tree a) where
  Void             == Void             = True
  (Branch x l1 r1) == (Branch y l2 r2) = (x==y) && (l1 == l2) && (r1==r2)
  _                == _                = False
```

a debe ser instancia de Eq para poder tener sobrecargado el operador == para valores de tipo a ya que se usa en la segunda ecuación.

Ejercicios

Consideremos el tipo Nat que hemos definido anteriormente. Se pide:

1. sobrecargar los operadores aritméticos (+ y *) de la clase Num para poder usarlos para sumar y multiplicar naturales
2. sobrecargar show de la clase Show para mostrar los naturales como el número que representan, por ejemplo Zero como 0, Suc Zero como 1, …
3. sobrecargar los métodos necesarios de Enum para poder definir listas aritméticas con valores de tipo Nat. Por ejemplo: [Zero..Suc (Suc Zero)] daría [Zero, Suc Zero, Suc (Suc Zero)]
4. Sobrecargar < de la clase Ord para poder comparar valores del tipo Figure según el valor de su área
5. Sobrecargar show de la clase Show para visualizar un círculo con su radio entre paréntesis (e.g., (2.5)) y un rectángulo con sus lados entre corchetes separados por una coma (e.g., [1.5,2.5]).