Paradigmas de Programación

6. Paradigma Funcional

Departamento de Informática
Universidad de Valladolid

Curso 2023-24

Grado en Ingeniería Informática
Grado en Estadística

CONCEPTOS FUNDAMENTALES

Paradigma Funcional (puro)

- No existe operación de asignación.
- Las “variables” almacenan definiciones o referencias a expresiones.
- Basado en el concepto (matemático) de **función**.
- La operación fundamental es la aplicación de una función a una serie de argumentos. La evaluación se guia por el concepto de sustitución.
- Un programa consiste en una serie de definiciones (de funciones, de tipos de datos..)
- Las estructuras de control básicas (y generalmente únicas) son la composición y la recursión.

Objetivos:

- **Transparencia Referencial** : Las funciones no tienen efectos laterales y su resultado está determinado únicamente por los valores de sus parámetros.
  - No modifican sus parámetros
  - No acceden ni modifican variables o estado globales
- Con ello se consigue una mayor grado de modularidad en independencia.
- El análisis de la corrección de un programa es más sencillo (y puede usar técnicas más potentes).
- Se puede implementar concurrencia de manera más natural (incluso automatizar su aplicación)

Funciones

- En los lenguajes funcionales, las funciones son entidades de primer orden (high-order), con un estatus similar al de los valores:
  - Pueden pasarse como parámetros a otras funciones
  - Pueden ser devueltas por funciones
  - Pueden combinarse (composición, aplicación parcial, ..) con otras funciones para definir otras nuevas.
  - Tienen un tipo de datos asociado.
- Funciones y valores en Haskell:
  - Las funciones en Haskell están currificadas (máximo una entrada, una única salida)
  - Los valores se obtienen mediante constructores de datos, que pueden verse como funciones "congeladas".
  - ¡Todo es una función en Haskell!

Problemas:

- Interacción con el mundo exterior: Al realizar acciones de I/O (entrada/salida) es inevitable el producir efectos laterales.
- Eficiencia: Si no se pueden modificar datos es necesario crear duplicados que incorporen la modificación.
- Complejidad de programación: Si no existe un estado externo, se debe enviar a cada función todos los datos necesarios.
- Sistema de tipado: Es dificil imaginar cómo incorporar un sistema de tipado estricto y/o O.O. a un enfoque funcional.

El problema del mundo externo

- En un modelo funcional puro, las funciones **no** pueden tener **efectos laterales**:
  - La salida sólo puede depender de la entrada.
  - No se pueden modificar entidades (sólo crear otras nuevas y darlas como resultado)
  - Una función sin parámetros debe devolver siempre el mismo resultado (es igual a una constante).
  - No tiene sentido una función sin resultado.
- Esto provoca **problemas** conceptuales:
  - ¿Cómo definir un generador de números aleatorios?
  - ¿Cómo definir una función que devuelva la fecha/hora?
  - ¿Cómo definir una función que escriba en pantalla?
  - ¿Cómo definir una función que lea un dato del usuario?

Posibles soluciones

- **Abandonar la pureza**: Permitir que exista un subconjunto de funciones que tengan efectos laterales (Lisp, Scheme)
- **Introducir el mundo exterior como entidad**:
  - random(generador) : { generador' , valor aleatorio }
  - getClockTime(mundo) : fecha/hora
  - putStrLn(mundo , cadena) : mundo'
  - getLine(mundo) : { mundo' , cadena }
- Problemas de este enfoque: Mundos paralelos

  mundo2 = putStrLn(mundo1,"Hola")

  mundo3 = putStrLn(mundo2,"Mundo")

  mundo4 = putStrLn(mundo2,"Pascual")

Solución de Haskell: Mónadas

- **Mónadas**: Concepto de teoría de tipos (categorías). Permite encapsular las acciones impuras de manera que las funciones que las realicen sigan presentando una interfaz pura (sin efectos laterales) al exterior.
- La mónada IO (**IO α** en Haskell) representa una **acción** que, al ser evaluada, genera un efecto en el mundo exterior.
  - **putStrLn :: String → IO ()** : Devuelve una acción, que al ser evaluda genera el efecto de mostrar una cadena por pantalla
  - **getLine :: IO String** : Devuelve una acción (siempre la misma): La acción de pedir una cadena de texto al usuario. El valor introducido se encapsula dentro de la mónada que representa la acción.

¿Problemas de Eficiencia?

- En los lenguajes funcionales los valores son **inmutables**. Por ejemplo, al añadir un elemento a una lista, el resultado es **otra lista** (con los mismos elementos más el nuevo) distinta a la original (que no sufre cambios).
- Consecuencia de la transparencia referencial (no existen efectos laterales ni modificaciones)
- Parece evidente que esto tiene serias consecuencias respecto a la eficiencia (en este caso referente al espacio). Si insertamos *n* elementos en una lista, acabamos con *n* listas distintas ocupando un espacio total de $n^2 / 2$ elementos.
- No existen estructuras equivalentes a los arrays (acceso directo en tiempo constante).

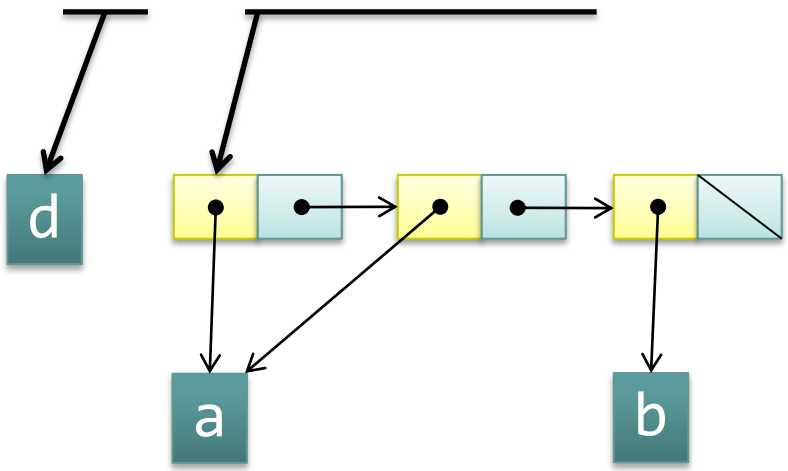
Si, pero no tan graves

- En general (y para problemas normales) los lenguajes funcionales son menos eficientes.
- Pero no de una forma tan drástica:
  - Las estructuras son enlazadas, y siempre se almacenan referencias a valores, no los propios valores.
  - Al ser inmutables, varias estructuras pueden compartir referencias a elementos o trozos enteros de otras.
  - Al no almacenar estado, si una estructura no va a ser usada en cálculos subsiguientes, se puede reciclar (su espacio se libera inmediatamente)
  - Suelen existir mecanismos (aunque bastante sofisticados) para simular el equivalente de los arrays (la mónada de estado en Haskell)

Ejemplo: Inserción en lista

- Supongamos la función insertar al principio que recibe un elemento y una lista y devuelve otra lista con el elemento añadido al principio:

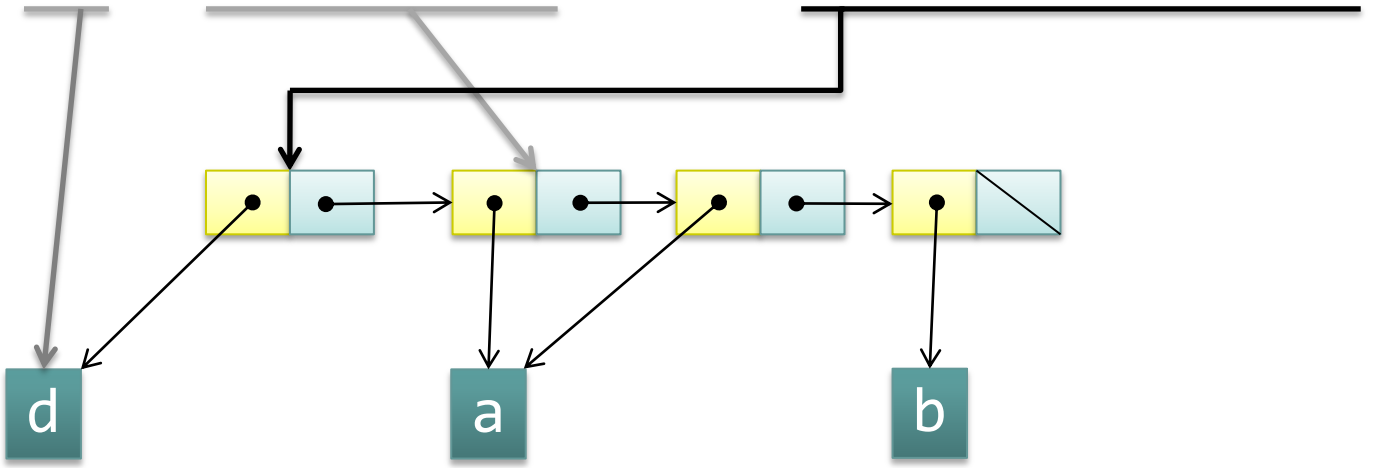
```
ins("d",["a","a","b"])
```



Ejemplo: Inserción en lista

- Supongamos la función insertar al principio que recibe un elemento y una lista y devuelve otra lista con el elemento añadido al principio:

```
ins("d",["a","a","b"]) → ["d","a","a","b"]
```



ELEMENTOS BÁSICOS DE HASKELL

Lenguaje Haskell

- Síntesis diseñada por expertos de la familia ML de lenguajes de programación (1990)
- Muy influyente (C#, Python, Scala, Ruby, ...)
- Es un lenguaje **funcional puro**.
  - Tipado Algebraico con inferencia de tipos.
  - Tipado estricto y seguro.
  - Funciones currificadas.
  - Concordancia de Patrones.
  - Evaluación perezosa/diferida.
  - Listas infinitas.
  - I/O y estilo pseudo-imperativo mediante Mónadas.

Entorno Haskell

- www.haskell.org (GHCi, Hughs, ..)
- Se puede elegir entre modo interpretado y modo compilado.
- El modo interpretado trabaja dentro de una mónada I/O
- Contenido típico de un programa Haskell:
  - Definiciones de tipos de datos
  - Definiciones de funciones con su signatura de tipo
  - Una función tiene el papel de punto inicial de ejecución (si se requiere interactividad se usa mónadas I/O)
  - Esa función se invoca desde el intérprete.

Estructura de un programa (I)


```
module Main where
  main :: IO()
  main = do
           putStr "Introduzca valor = "
           v <- readLn :: IO Integer
           putStrLn (show (fact v))

  -- Función factorial
  fact :: Integer -> Integer
  fact n = if n <= 1 then 1 else n * fact (n-1)

  -- Definición de nuevos tipos
  data Pila a = PilaVac | MkPila a (Pila a)

  -- Declaración de instancias de clase
  instance (Show a) => Show (Pila a) where
    show PilaVac     = ""
    show MkPila x p = show x ++ "," ++ show p

  -- Función insertar en pila
  insPila :: Pila a -> a -> Pila a
  insPila p x = MkPila x p
```

Función principal (Síntaxis de mónada)

Definición de funciones

Nuevos tipos de datos

Instancia de clase

Definición de funciones

Estructura de un programa (II)


```
module Main where
  main :: IO()
  main = do
           putStr "Introduzca valor = "
           v <- readLn :: IO Integer
           putStrLn (show (fact v))

  -- Función factorial
  fact :: Integer -> Integer
  fact n = if n <= 1 then 1 else n * fact (n-1)

  -- Definición de nuevos tipos
  data Pila a = PilaVac | MkPila a (Pila a)

  -- Declaración de instancias de clase
  instance (Show a) => Show (Pila a) where
    show PilaVac     = ""
    show MkPila x p = show x ++ "," ++ show p

  -- Función insertar en pila
  insPila :: Pila a -> a -> Pila a
  insPila p x = MkPila x p
```

Declaración de tipo

Definición de función

Constructores de valores

Restricciones de tipos

Concordancia de patrones

Genericidad

Valores, Tipos predefinidos

- Haskell tiene los tipos simples tradicionales:
  - tipo `Int` → Números enteros
  - tipo `Integer` → Enteros de tamaño arbitrario
  - tipo `Double` → Números reales
  - tipo `Char` → Caracteres (entre comillas simples: `'a'`, `'/n'`)
  - tipo `Bool` → Valores lógicos (literales `True` y `False`)
- Tipos compuestos:
  - tipo `[a]` → Listas que contienen elementos de tipo `a` (todos los elementos deben ser del mismo tipo). Ejemplos de listas: `[1,2,3]` ≡ `1:[2,3]` ≡ `1:2:3:[]`.
  - tipo `(a,b,..)` → Tuplas. Pueden contener elementos de tipos distintos. Ej. `('x',True,2)`
  - tipo `String` → Cadenas de caracteres (entre comillas dobles, ej. “Hola”). Es una lista de caracteres: `String` ≡ `[Char]`

Tipos Funcionales

- Las funciones tienen asociado un tipo de datos.
- El operador `->` se usa en las declaraciones de tipos para indicar un tipo funcional.
- `a -> b` es el tipo de las funciones que reciben un parámetro de tipo `a` y devuelven un resultado de tipo `b`
  - Es asociativo a la derecha:

    `a -> b -> c -> d ≡ a -> (b -> (c -> d))`

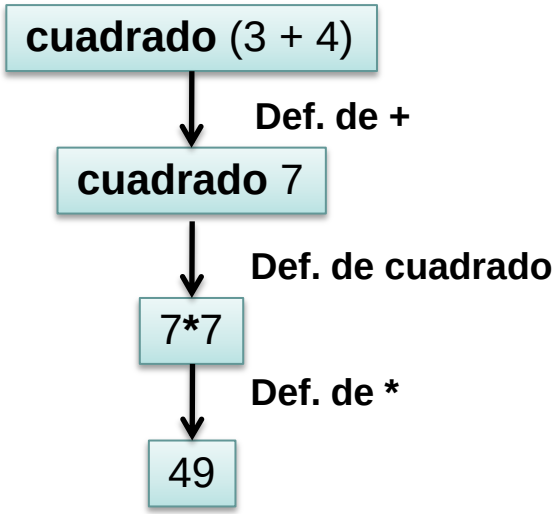
  - El tipo anterior denota una función que toma un valor de tipo `a` y devuelve otra función, la cual toma un valor de tipo `b` y devuelve otra función, la cual toma un valor de tipo `c` y devuelve un valor de tipo `d`.
  - Para no liarse, se puede suponer también que es una función con tres parámetros de tipos `a`, `b` y `c` y devuelve un valor de tipo `d`

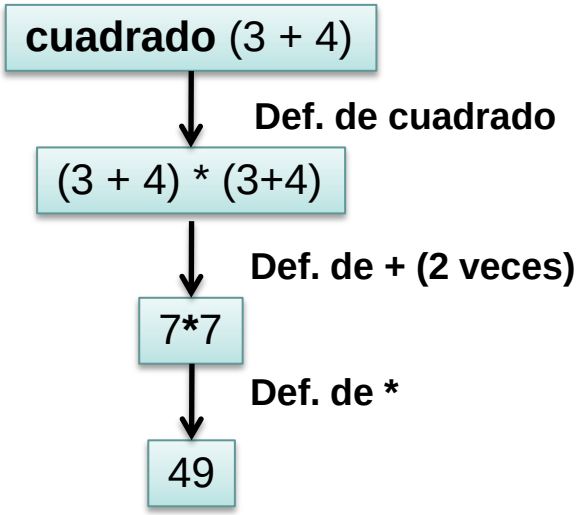
Declaraciones de tipo

- Las declaraciones de tipo se usan para indicar el tipo de un valor o de una función.
- El símbolo `::` se traduce como "*pertenece al tipo*".
  - `x :: Int` indica que el identificador x hace referencia a un valor entero (o bien, se puede ver como una función sin parámetros y resultado entero).
  - `raiz :: Double -> Double` indica que el identificador raiz hace referencia a una función que recibe un valor real y devuelve un valor real.
  - `max :: Int -> Int -> Int` indica que max es una función que recibe un entero y devuelve una función que recibe un entero y devuelve un entero (o bien, se puede pensar en una función que recibe dos enteros y devuelve un entero).

Definición de funciones

- La definición de una función en Haskell indica la expresión por la que se puede sustituir una aplicación de la función a un parámetro (una "llamada" a la función) al evaluar una expresión en la que aparezca.
  - Dada la función elevar al cuadrado: **cuadrado** x **=** x*****x
  - La expresión `cuadrado(3+4)` se puede evaluar de dos formas:





Evaluación de expresiones (I)

- La evaluación directa implica evaluar primero las subexpresiones (los parámetros) de una expresión antes de calcularla (evaluar primero los parámetros antes que la función)
  - Es el método normal de evaluación en lenguajes imperativos (y en Scheme)
  - Ejemplo: En Scheme el código siguiente (calcular el recíproco) da error si x = 0:

```
(cond (= x 0) 1 (/ 1 x))
```

- La evaluación diferida o perezosa implica evaluar primero la expresión (y sólo se evalua una expresión si se necesita el valor).

Evaluación de expresiones (II)

- Haskell utiliza el modelo de evaluación diferida:
  - Al evaluar una expresión (lo que siempre implica la evaluación de una función --> la aplicación de parámetros) el compilador recorre las definiciones de esa función en el orden en que aparecen en el código.
  - Cuando encuentra una definición que concuerda (concordancia de patrones), sustituye la definición por la expresión, y detiene la búsqueda.
  - Cuando se obtiene el resultado, se detiene la evaluación (aunque existan subexpresiones que todavía pudieran ser evaluadas)
  - Existen expresiones que no se evalúan (Formas normales): Los datos y las funciones constructoras.

Funciones: Aplicación

- En Haskell la aplicación de parámetros a una función se realiza mediante la mera yustapoxición:

```
max :: Int -> Int -> Int
max x y = if x > y then x else y
```

  - Para evaluar el máximo de 3 y 7 se escribe: `max 3 7`
  - Y no `max(3,7)` (daría un error de compilación ya que entiende que se pasa una tupla de enteros)
  - El motivo es conseguir una sintaxis que pueda adaptarse facilmente a casos más generales (parámetros funciones)
  - La aplicación de una función tiene precedencia máxima:

```
max 3 7 ≡ (max 3) 7
```

Uso de funciones

- Por ejemplo, supongamos que queremos definir un operador para representar la composición de funciones, el operador punto: $(f \cdot g)\, x = f\,(g\,(x))$
- Este operador recibe dos funciones y devuelve la función resultado de su composición. Supondremos que ***x*** es de tipo ***a***, la función ***g*** lo transforma en un valor de tipo ***b*** y la función ***f*** toma ese valor y devuelve uno de tipo ***c***. El resultado será una función que recibe un valor de tipo ***a*** y devuelve un valor de tipo ***c***.

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

Funciones y operadores

- Los operadores de Haskell son todos binarios.
- No existe distinción entre operadores y funciones: Los operadores son funciones (binarias) que se aplican con una sintaxis especial.
- Todo operador puede actuar como función (basta escribirlo entre paréntesis) `(+) 3 7`
- Toda función (binaria) puede actuar como operador (basta con escribirlo entre acentos) `3 `max` 7`
- Existen formas de definir nuevos operadores y de indicar su precedencia y asociatividad.

Secciones (I)

- La función máximo en realidad no tiene dos parámetros: Sólo tiene uno, y devuelve una función.

```
max :: Int -> (Int -> Int)
max x y = if x > y then x else y
```

Paréntesis implícitos !

- Se puede definir una una función "devuelve el argumento o 10 si es más pequeño" usando max:

```
max10 :: Int -> Int
max10 y = max 10
```

- Con los operadores se puede hacer lo mismo, obteniendo lo que se denomina sección.

```
(+1) -- Función incremento
```

Secciones (II)

- Las secciones son una forma de obtener nuevas funciones fijando algún parámetro de otra función ya existente:

```
(+1) -- Función incremento
```

```
(*2) -- Función duplicar
```

```
(1/) -- Función recíproco
```

```
(^2) -- Función cuadrado
```

```
(<3) -- Función "es menor que 3"
```

```
(+(-1)) -- Función decremento. (-1) se interpreta como
           el entero -1.
```

Definición de funciones

- Expresiones Condicionales:

```
fact n = if n > 1 then 1 else n * fact (n-1)
```

```
signo n = if n == 0 then 0
          else if n > 0 then 1 else -1
```

- Guardas:

```
fact n
  | n > 1     = 1
  | otherwise = n*fact (n-1)
```

```
signo n
  | n == 0 = 0
  | n > 0  = 1
  | n < 0  = -1
```

Concordancia de Patrones

- Esta técnica permite escribir funciones usando varias definiciones en las cuales en vez de parámetros se indican patrones (mezcla de valores y variables):
  - Si los valores concuerdan con los parámetros, se usará esa definición.
  - Si es posible instanciar la(s) variable(s) con valores que concuerden, se usará esa definición y dentro de ella las variables tendrán el valor adecuado.
  - Un guión bajo indica un patrón que siempre se cumple.
  - Las definiciones se examinan en dirección descendente (y se detiene la búsqueda cuando una concuerda):

```
fact 0 = 1
fact n = n*fact (n-1)
```

```
or :: Bool -> Bool -> Bool
or True _ = True
or False x = x
```

Subexpresiones (I)

- Es posible evitar el tener que calcular varias veces una subexpresión utilizando cláusulas where.

```
-- Una Solución de una ecuación de 2º grado
eq2grad :: Double -> Double -> Double -> Double
eq2grad a b c
  | d >= 0     = (-b + sqrt(d))/(2*a)
  | otherwise  = error "Valores complejos"
  where
    d = b*b - 4*a*c
```

- Podemos pensar en la variable d como una "subfunción" definida dentro de eq2grad. De hecho se pueden definir funciones cualesquiera (no sólo constantes) en cláusulas `where`.

Subexpresiones (II)

- Es posible definir varias variables y hacerlas depender de otras definidas anteriormente:

```
-- Cadena con soluciones de una ecuación de 2º grado
eq2grad :: Double -> Double -> Double -> String
eq2grad a b c
  | d >= 0    = show v1 ++ ", " ++ show v2
  | otherwise = "Valores complejos"
  where
    d = b*b - 4*a*c
    r = if d >= 0 then sqrt d else 0
    a2 = 2*a
    v1 = (-b + r)/a2
    v2 = (-b - r)/a2
```

Operadores y funciones

Operando	Argumentos	Resultado
`+ - *`	Numericos (mismo tipo)	Operación aritmetica (mismo tipo)
a `^` *b*	*a* numérico, *b* entero	Potencia (mismo tipo que *a*)
a `**` *b*	*a* real, *b* real	Potencia (mismo tipo)
`div mod`	Enteros	Cociente y resto (mismo tipo)
`/`	Reales	División real (mismo tipo)
`&& || not`	Booleanos	And, or, not (booleano)
`== /=`	Equivalentes (clase **Eq**)	Igual, distino (booleano)
`< <= > >=`	Comparables (clase **Ord**)	Menor, menor-igual, ...
f `.` *g*	Dos funciones	La función *f(g(x))* [Composición]
f `$` *x*	Una función y un valor	El valor *f(x)* [Aplicación]
`pred, succ`	Enumerable (clase **Enum**)	El valor anterior / siguiente (m.t.)
`show`	Traducible (clase **Show**)	Traduce a String el valor

Ejemplo: Test de primalidad (I)

- Vamos a crear un función que detecte si un valor n es primo comprobando si es divisible por alguno de los valores $2..\sqrt{n}$. Primero la versión imperativa:

```
static boolean esDiv(int a, int b) {
    return a % b == 0;
}

static boolean esPrimo1(int n) {
    if(n < 2) return false;
    if(n == 2) return true;
    int d = 2;
    while (d*d <= n && !esDiv(n,d)) {
        d++;
    }
    return d*d > n;
}
```

Ejemplo: Test de primalidad (II)

- El bucle de comprobación de divisores se codifica recursivamente, usando una función local:

```
-- Test de divisibilidad, para clarificar el código
esDiv :: Int -> Int -> Bool
esDiv a b = a `mod` b == 0

esPrimo1 :: Int -> Bool
esPrimo1 n
  | n < 2     = False
  | n == 2    = True
  | otherwise = not (test n 2)
  where
    test n d = if d*d > n
               then False
               else esDiv n d || test n (d+1)
```

SISTEMA DE TIPADO

Genericidad restringida (I)

- Ademas de tipos de datos existen **clases** de tipos (no tiene que ver con orientación a objetos).
- Una clase es un conjunto de tipos para los cuales está garantizado que existen una serie de funciones (con nombre y signatura definida).
- Si un tipo pertenece a una clase (es una **instancia** de la clase) entonces se sabe que tiene definidas las funciones de la clase.
- Para incluir un nuevo tipo (definido por usuario) en una clase, se debe definir las funciones de esa clase.

Genericidad restringida (II)

- Por ejemplo, la clase **Num** en Haskell contiene a los tipos de datos que representan números, y tiene la definición:

Pseudo-herencia de clases !

```
class (Eq a, Show a) => Num a where
  (+), (-), (*) :: a -> a -> a
  negate :: a -> a
  abs, signum :: a -> a
  fromInteger :: Integer -> a
```

- Significado: Todo tipo a que pertenezca a Num debe pertenecer también a las clases Eq y Show, y definir las operaciones: Suma, Resta, Producto, Negación, Valor Absoluto, Signo (todas con parámetros y resultado de tipo a) y Traducción de un entero a un valor tipo a

Genericidad restringida (III)

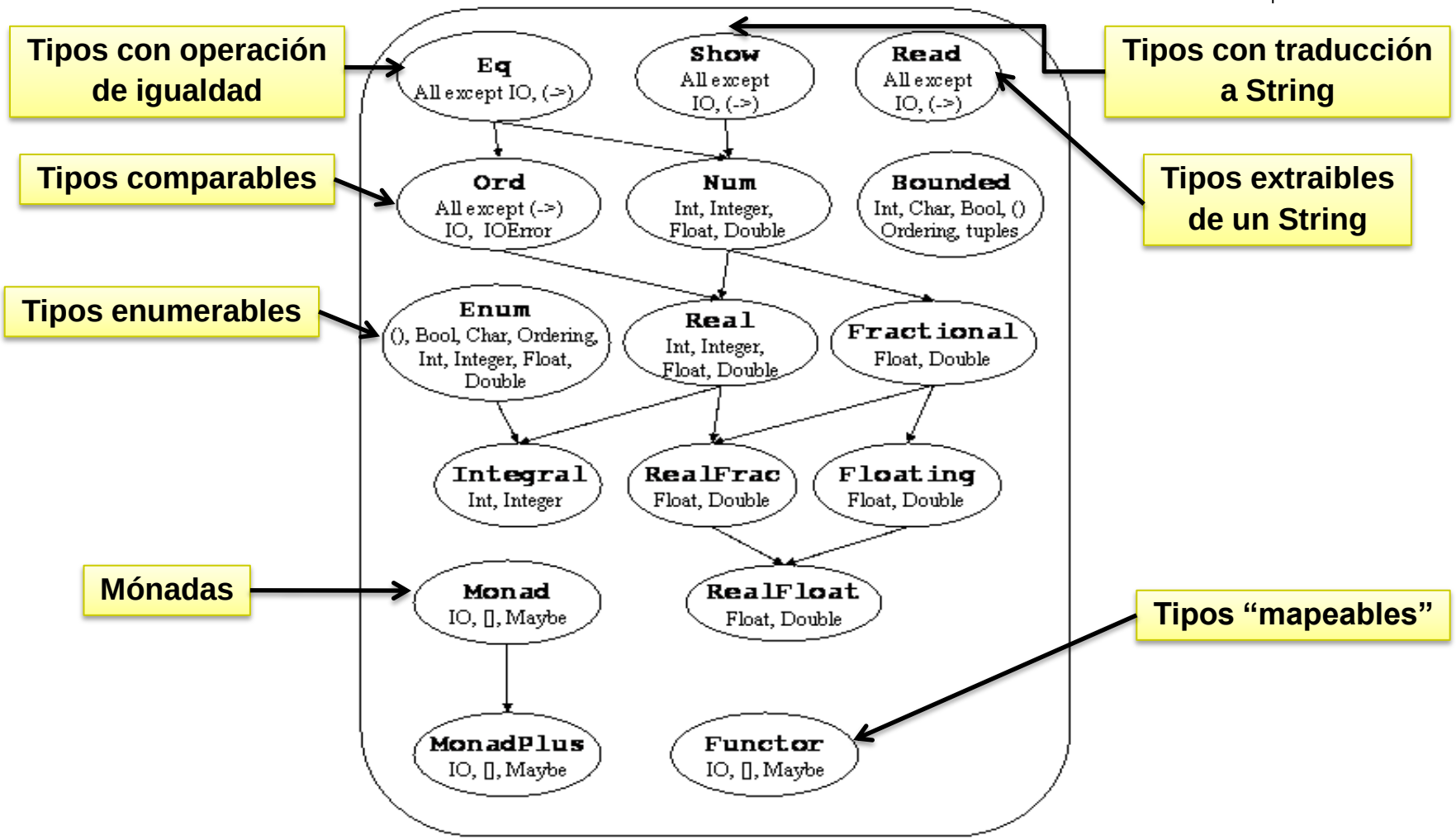
- La notación `(Clases,..) => Declaración` significa que el tipo de datos debe pertenecer a la clase.
- Por ejemplo a la clase Num pertenecen los tipos Int, Integer y Double. De esta forma se pueden escribir funciones que trabajen con cualquiera de esos tipos de datos (u otros nuevos que se hagan instancia de Num)
- La siguiente función se puede usar con entradas de tipo Int, Integer o Double, ya que existen las versiones adecuadas de (*) y (+) para ellas:

```
sum_pot :: (Num a) => a -> a -> a
sum_pot x y = x*x + y*y
```

Genericidad restringida (IV)

- Las clases tienen una cierta semejanza con las interfaces o clases abstractas de OOP, pero mientras que en OOP los objetos “llevan consigo” los métodos para trabajar con ellos..
- .. en Haskell es el sistema el que mantiene una especie de tabla de funciones “sobrecargadas” y selecciona la adecuada dependiendo del tipo.
- Las clases tienen una especie de “herencia”: La posibilidad de exigir la pertenencia a otras clases.
- El programador puede definir nuevas clases y declarar tipos como pertenecientes a una clase (indicando las funciones apropiadas).

Clases estándar de Haskell



Sistema de Tipado

- El sistema de tipado es **estricto y seguro**.
  - Todo valor pertenece a un tipo de datos
  - Toda función pertenece a un tipo de datos (tipo funcional)
  - Para toda expresión se puede saber, sin ambiguedad, el tipo de datos resultado.
- Los errores de tipo se detectan en tiempo de compilación.
- Haskell tiene **inferencia de tipos**: No es necesario (aunque si conveniente) declarar el tipo de ningún elemento (valor, función, expresión), el compilador puede averiguarlo (inferencia Hildner-Millner).
- Esta caracteristica permite definir expresiones y funciones locales sin necesidad de declararlas.

Tipado Algebraico

- Haskell usa un sistema de tipado algebraico
- Todo valor proviene de un constructor de datos
- Los constructores son (¡sorpresa!) funciones:
  - No evaluables: Formas normales. Se puede pensar en ellas como en etiquetas que identifican a los valores.
  - Función sin parámetro ➔ Valor constante.
  - Función con parámetros ➔ Encapsula varios datos en los parámetros: Se puede considerar como un registro.
  - Se usa la concordancia de patrones para acceder a esos parámetros (los "campos" del "registro")
- Se pueden definir varios constructores para un mismo tipo de dato (tipos union, similar a variantes)
- Los tipos de datos pueden ser recursivos

Definición de tipos

- Una definición de tipos tiene la sintaxis: (las llaves indican repetir 0+ veces):

```
data Tipo = Constructor { | Constructor }
```

- Donde cada constructor consiste en:

```
Nombre { Tipo_Argumento }
```

- El significado es que los valores de ese tipo se pueden obtener mediante alguna de las funciones constructoras que se listan (separadas por |), y que esas funciones pueden ir acompañadas de 0 o más valores (los parámetros) que tienen los tipos especificados.

Ejemplos (I)

- Queremos representar a una persona por su nombre, su edad y su género:

```
data Genero = Mujer | Hombre | Desconocido
data Persona = Jefe | Cliente String Int Genero
```

- En violeta aparecen las funciones constructoras.
  - `Mujer`, `Hombre` y `Desconocido` son funciones sin argumentos que construyen un valor de tipo `Genero`
  - Un valor de tipo `Persona` se puede obtener por el constructor `Jefe` o por el constructor `Cliente`, el cuál es una función con tres argumentos (de tipos string, entero y género)
- Ejemplo de valor de tipo Persona

```
Cliente "Chus" 45 Desconocido
```

Ejemplos (II)

- Para acceder a los datos se utiliza concordancia de patrones. Por ejemplo, una función que devuelva el saludo adecuado para un portero de discoteca automatizado:

```
saludo :: Persona -> String
saludo Jefe = "Todo en orden."
saludo Cliente "CVR" _ _ = "Su suit está preparada"
saludo Cliente nom edad gen
  | edad < 20 = "p'alante"
  | otherwise = case gen of
                  Mujer -> "Señora " ++ nom
                  Hombre -> "Señor " ++ nom
                  Desconocido -> "Hola, " ++ nom
```

- **Nota**: Las estructuras case permiten usar concordancia de patrones en expresiones.

Tipos recursivos y paramétricos

- Se puede usar el propio tipo de datos que se está definiendo como tipo de parámetro de un constructor.
- Se parametrizar el tipo respecto a otros, indeterminados, sobre los que se pueden poner restricciones (pertenencia a una clase)
- Por ejemplo, una lista puede definirse como una lista vacía o bien un elemento seguido del resto de la lista:

```
data Lista a = Vacia | Nodo a (Lista a)
```

- En este caso el tipo **`Lista`** depende de otro (indicado por **`a`**), y el constructor **`Nodo`** contiene dos argumentos, uno de tipo **`a`** (el primer valor) y una lista de **`a`**'s (el resto de la lista).

Listas como tipo recursivo

- Una lista que contenga los números 1,2,3 se indicaría:

```
Nodo 1 (Nodo 2 (Nodo 3 Vacia))
```

- Y aquí el tipo **a** sería **Integer**. En la siguiente lista el tipo **a** sería **String**:

```
Nodo "Soy" (Nodo "una" (Nodo "lista" Vacia))
```

- El tipo lista de Haskell se define de esta forma, tomando el constructor `[]` el lugar de `Vacia` y el operador `:` el lugar de `Nodo`:

```
"Soy":("una":("lista":[]))
```

- Dando asociatividad derecha a `:` tenemos

```
"Soy":"una":"lista":[] ≡ ["Soy","una","lista"]
```

Tipos predefinidos

- Todos los tipos predefinidos de Haskell siguen el mismo esquema, aunque algunos tienen una sintaxis especial para facilitar su uso (ej. listas, tuplas)

```
data () = ()                              -- Tipo nulo
data Bool = False | True                  -- Booleanos
data Char = .. |'a'|'b'|'c' ..            -- Caracteres
data Int = .. |-1| 0| 1| 2 ..             -- Enteros
data Ordering = LT | EQ | GT              -- Res. ordenación
data [a] = [] | a : [a]                   -- Listas
data (a,b) = (a,b)                        -- Tuplas
data Maybe a = Nothing | Just a           -- Nulificables
data Either a b = Left a | Right b -- Alternativa

type String = [Char]  -- Ejemplo de tipo sinónimo
```

LISTAS
MAP FILTER FOLDER

Listas

- Las listas son la estructura de datos básica en Haskell (y en la mayoría de lenguajes funcionales).
- No solo sirven para almacenar datos, sino que son un elemento fundamental en el diseño de algoritmos.
- Una lista es un par `(x : xs)` donde `x` es el primer elemento de la lista y `xs` la (sub)lista que contiene el resto de los elementos.
- La lista vacía se indica por el símbolo `[]`. No concuerda con el patrón `(x:xs)`. Si tenemos una lista de un elemento, `[z]`, si concuerda y `x ← z` y `xs ← []`
- En Haskell la sublista puede estar indicada por una expresión y por el mecanismo de evaluación diferida esto posibilita el poder definir (y usar) listas infinitas.

Listas: Funciones predefinidas

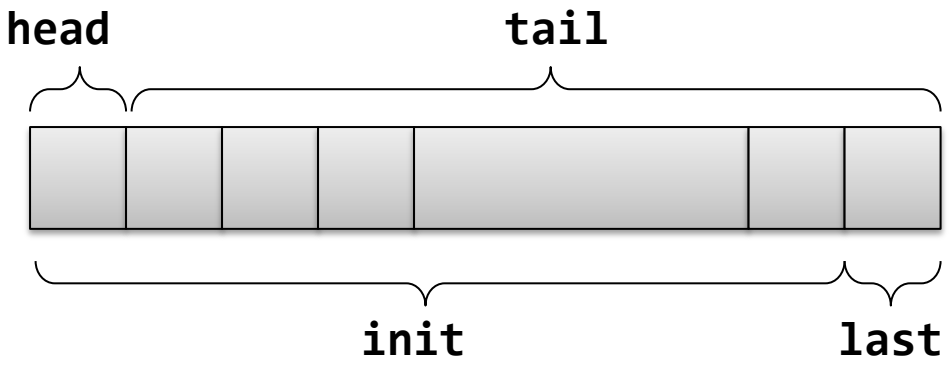
- Acceso a primer (head) y último (last) elemento, lista sin primer elemento (tail) y sin último elemento (init):

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

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

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

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



Funciones predefinidas (II)

- Longitud (length), elemento i-ésimo (!!), concatenar (++):

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

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

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

Funciones predefinidas (III)

- Sublistas: Obtener primeros *n* elementos (take), quitar primeros *n* elementos, obtener y quitar mientras se cumpla una condición:

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

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

Ejercicios: Obtener las correspondientes funciones **drop** y **dropWhile** (devuelven la lista tras quitar los correspondientes primeros elementos) y la función **reverse** (invierte una lista)

Nuevos elementos sintácticos

- La función predefinida **span** toma un predicado (una función booleana sobre los elementos de la lista) y devuelve dos listas, la de los primeros elementos mientras cumplen la condición y el resto de la lista
- **`span p lis = (takeWhile p lis , dropWhile p lis)`**

```
span :: (a -> Bool) -> [a] -> ([a],[a])
span p [] = ([],[])
span p lis@(x:xs)
          | p x       =  (x:ys,zs)
          | otherwise =  ([],lis)
          where (ys,zs) = span p xs
```

Se devuelve una tupla de listas

Nombre global de patrón

Concordancia de patrones en where

Funciones anónimas

- En Haskell (y Python) se pueden definir funciones anónimas (tambien llamadas abstracciones lambda)
  - No tienen nombre, tan sólo indican los parámetros que necesitan y el valor que devuelven.
  - El objetivo es usarlas en aquellos puntos donde necesitamos una función sencilla y no queremos molestarnos en definirla aparte.
  - Pueden usar variables del ámbito en que se usan
  - Se puede usar concordancia de patrones en sus parámetros
- Sintaxis (los paréntesis sólo si se necesitan):

```
(\(param {,param}) -> expresion)
```

Ejemplo

- Tenemos un texto (en forma de String → lista de caracteres) con frases terminadas en punto. Queremos obtener una lista de las frases que lo componen:
- Podemos usar **span** indicando como condición extraer caracteres mientras sean distintos del punto. La condición se puede escribir como una función anónima:

```
span (\c -> c /= '.') "Si. No. Quizas." →
("Si",". No. Quizas.")
```

- Intente resolverlo por si mismo. La solución consiste en usar **span,** el cual devuelve una tupla, de la cual hay que extraer el primer elemento, añadirlo a la lista que generamos, y recursivamente generar el resto de la lista aplicando **span** al resto (segundo elemento de la tupla) pero con cuidado de quitar el punto (si no **span** se atasca).

Solución

- El problema es que todo eso hay que escribirlo en una única expresión, sin secuencias. Para resolverlo debemos usar cláusulas where aprovechando que se puede usar concordancia de patrones en ellas (así podemos "extraer" y nombrar el resultado de **span)**

```
frases :: String -> [String]
frases "" = []
frases txt
  | resto == "" = [frase]
  | otherwise   = frase : frases (tail resto)
  where (frase,resto) = span (\c -> c /= '.') txt
```

Listas enumeradas (rangos)

- Todo tipo de datos que pertenezca a la clase **`Enum`** tiene definidas funciones (**`enumFrom`**, etc.) para generar rangos de valores. Existe una sintaxis especial para crear listas basadas en esos rangos:
  - **`[1..10]` → `[1,2,3,4,5,6,7,8,9,10]`**
  - **`['a'..'z']` → `"abcdefghijklmnopqrstuvwxyz"`**
  - **`[1,3..10]` → `[1,3,5,7,9]`**
  - **`[1..]` → `[1,2,3,...]` Lista infinita!**
  - Es posible crear listas infinitas porque Haskell tiene evaluación diferida. Si una operación sólo trabaja sobre una parte de la lista, no existe ningún problema:

```
take 5 [1..] → [1,2,3,4,5] (Correcto)
length [1..] → ... (Nunca termina)
```

Map, Filter, Folder

- Existe un grupo de funciones sobre listas (tipicamente se denomina map-filter-reduce) que se han mostrado extremadamente útiles en la resolución de gran cantidad de problemas:
  - **Map** aplica una misma operación (unaria) sobre todos los elementos de una lista, devolviendo la lista modificada.
  - **Filter** crea otra lista extrayendo sólo los elementos que cumplan una condición.
  - **Folder** (Reduce) "colapsa" una lista aplicando una operación binaria que consume un elemento más y el resultado del colapso actual.
- Otras funciones:
  - **Zip**, **ZipWith** combinan dos listas en una sola (lista de pares o lista basada en operaciones sobre los pares)

Map

- Recibe una función que transforma valores de tipo a en valores de tipo b y una lista de a's, y devuelve una lista de b's: La lista original aplicando la función a todos los valores.

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

- Ejemplos (**show** es la función que traduce un valor a cadena):

```
map (+1) [1,2,3,4] → [2,3,4,5]
map (show) [1,2,3,4] → ["1","2","3","4"]
map (\(x,y) -> x+y) [(1,2),(3,4)] → [3,7]
```

Ejemplos serios con Map

- Si intentamos aplicar **map** directamente a una matriz (una lista de listas) obtenemos un error:

```
map (+1) [[1,2],[3,4],[5,6]] ← Error
```

- El problema es que los elementos de la lista son a su vez listas, y no tiene sentido el sumar 1 a una lista.

- Lo que si podemos aplicar a cada elemento (lista) es la operación de aplicar (map) el incremento:

```
map (map (+1)) [[1,2],[3,4],[5,6]] →
[[2,3],[4,5],[6,7]]
```

- Ejercicio: Usar **map** para transponer una matriz

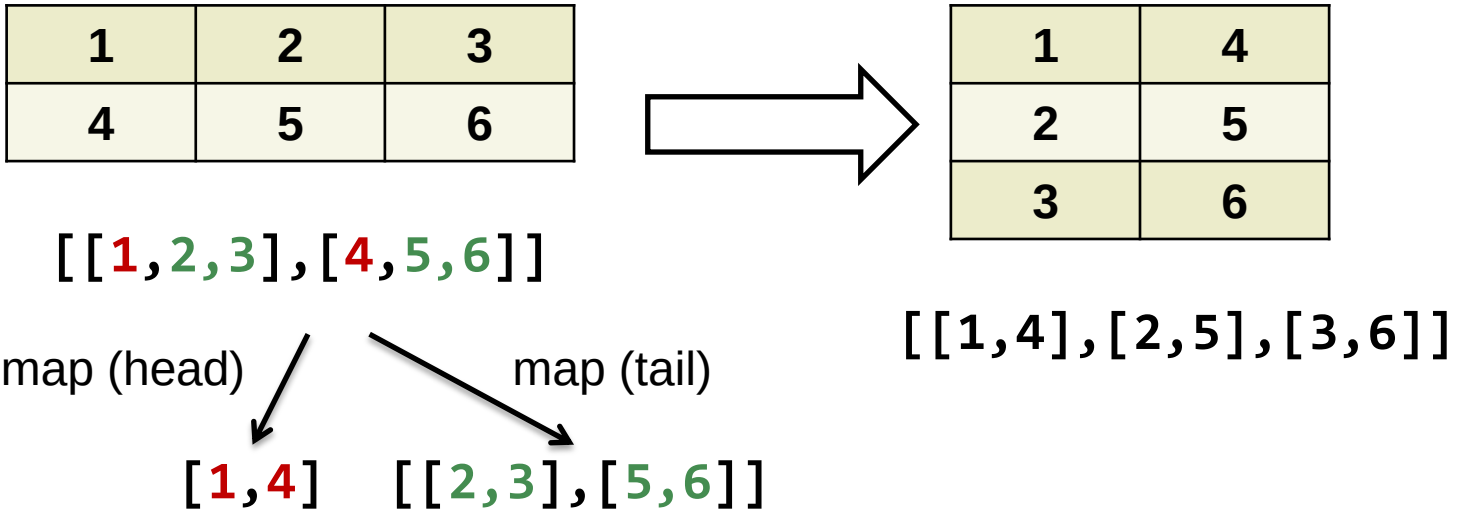
Transponer matriz (I)

- El transponer una matriz es la típica operación que se realiza de forma fácil en el modelo imperativo y es (aparentemente) muy complicada en el funcional:

```
procedure Transponer(const M: array[1..M,1..N] of ..;
                     var T: array[1..N,1..M] of ..);
var i,j : integer;
begin
  for i := 1 to N do
    for j := 1 to M do
      T[i,j] := M[j,i]
end;
```

- En el paradigma funcional se debe usar recursividad en vez de bucles, e ir construyendo la matriz (lista de listas) resultante con los resultados de las llamadas.

Transponer matriz (II)




```
transponer :: [[a]] -> [[a]]
transponer []       = []
transponer ([] : _) = []
transponer m = (map head m) : transponer (map tail m)
```

Filter

- Recibe un predicado y una lista de valores y devuelve la lista a la que sólo pertenecen los valores que cumplan el predicado:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) = if p x
                  then x : filter p xs
                  else filter p xs
```

- Ejemplo, lista infinita de primos. (usando la función **`esPrimo1`** de la transparencia 36):

```
primos :: [Int]
primos = filter esPrimo1 [1..]
take 8 primos → [2,3,5,7,11,13,17,19]
```

Filter - Ejemplos

- Función que devuelve el número de veces que se encuentra un elemento en una lista:

```
veces :: (Eq a) => [a] -> a -> Int
veces lis x = length (filter (==x) lis)
```

- Lista de primos menores que lim. Usamos el método de la Criba de Eratóstenes (filtrar múltiplos de primos):

```
primos :: Int -> [Int]
primos lim = criba (2:[3,5..lim])

criba :: [Int] -> [Int]
criba [] = []
criba (p:ps) = p : criba (
                    filter (\n -> n `mod` p /= 0) ps
                )
```

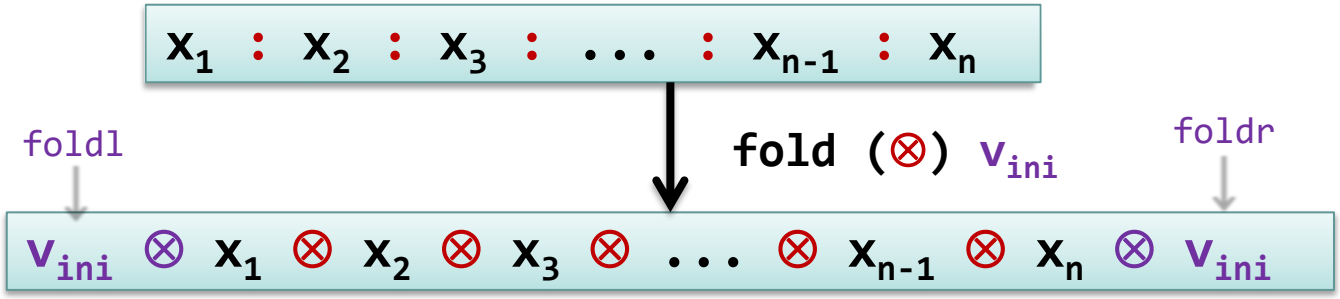
Lista de impares

p es primo

filtramos los múltiplos de p

Folder – Reduce

- Folder colapsa una lista de **a**'s en un único valor de tipo **b** (que tipicamente suele ser **a**). Para ello va consumiendo elementos de la lista y les aplica un operador binario junto con un valor acumulado, que al final será el resultado.
  - Existen variantes que depende de como se consuman los elementos (izda → dcha o al revés) y si se da un valor inicial o no.



  - Puede ser util visualizarlo como la sustitución del operador constructor de listas, **:** , por el operador (lo denotamos como ⊗) indicado.

Folder (I)

- Versiones izquierda:

$$((..(((v_{ini} \otimes x_1) \otimes x_2) \otimes x_3) \otimes ... \otimes x_{n-1}) \otimes x_n)$$

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

$$((..((x_1 \otimes x_2) \otimes x_3) \otimes ... \otimes x_{n-1}) \otimes x_n)$$

```
foldl1 :: (a -> a -> a) -> [a] -> a
foldl1 f (x:xs) =  foldl f x xs
```

```
foldl1 (-) [1,2,3,4] → ((1-2)-3)-4 = -8
```

Folder (II)

- Versiones derecha:

$(x_1 \otimes (x_2 \otimes (... \otimes (x_{n-1} \otimes (x_n \otimes v_{ini}))..)))$

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

$(x_1 \otimes (x_2 \otimes (... \otimes (x_{n-1} \otimes x_n)..)))$

```
foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 f [x] = x
foldr1 f (x:xs) =  f x (foldr1 f xs)
```

```
foldr1 (-) [1,2,3,4] → 1-(2-(3-4)) = -2
```

Ejemplos con Folder

- Suma de los elementos de una lista:

```
suma :: (Num a) => [a] -> a
suma lis = foldl (+) 0 lis
```

- Comprobar si un elemento pertenece a una lista:

```
existe :: (Eq a) => a -> [a] -> Bool
existe x lis = foldl1 (||) (map (==x) lis)
```

- Invertir una lista:

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

invertir :: [a] -> [a]
invertir = foldl (flip (:)) []
```

Zip y ZipWith

- **Zip** combina dos listas (con elementos de tipos **a** y **b**) en una lista de tuplas (si una lista es más larga, se ignoran los elementos sobrantes):

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

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

Constructor de tuplas

- **ZipWith** recibe una función para combinar los elementos generando valores de tipo **c**. Devuelve, en lugar de una lista de tuplas, una lista de valores tipo **c**.

Ejemplos con Zip

- Devolver la lista de índices en los que aparece un determinado valor:
  - Primero emparejamos cada elemento con su índice, usando zip y la lista infinita [1..] (recordad que zip se detiene cuando se acaba una lista)
  - Luego filtramos todos los pares cuyo elemento sea distinto del que buscamos.
  - Por último extraemos el índice del par.

Operador **aplicación** (para ahorrarnos escribir paréntesis)

```
busqueda :: (Eq a) => a -> [a] -> [Int]
busqueda x lis = map (\(_,i) -> i) $
                 filter (\(y,_) -> y == x) $
                 zip lis [1..]
```

Listas Infinitas

- Una lista puede tener en una parte definida por una expresión en la que aparezca la propia lista.
- Ejemplo: Secuencia de Fibonacci. En **fibs2** se define la lista como la suma de elementos entre ella y una versión de si misma desplazada a la izquierda:

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

fibs1 :: [Int]
fibs1 = map (fib) [1..]

fibs2 :: [Int]
fibs2 = 1:1:zipWith (+) fibs2 (tail fibs2)
```

```
fibs2:          [1,1,2,3,5,8,13,21,...]
tail fibs2:     [1,2,3,5,8,13,21,...]
-----------------------------------
zipWith (+)     [2,3,5,8,13,21,...]
```

Compresión de Listas

- Existe una sintaxis especial para representar listas definidas por compresión (en vez de por extensión, listando sus elementos) de una forma muy parecida a la notación matemática para definir conjuntos.
- Se puede contemplar como una combinación de **map** y **filter** (aunque más potente)
- Sintaxis:

```
[expr | {var <- lista, condición}]
```

generador

- Significado:
  - Construye la lista formada por elementos obtenidos de evaluar la expresión
  - La expresión se evalua para todos los valores de los generadores, salvo los filtrados por la condición
  - Cualquier expresión puede usar elementos definidos en elementos situados a la derecha.

Ejemplos: Compresión de Listas

- Los tripletes pitagóricos son los trios (x,y,z) que cumplen la condición $x^2+y^2 = z^2$. Queremos listar todos los tripletes que cumplan $x < y < z < lím$.

```
tripletes :: Int -> [(Int,Int,Int)]
tripletes lim = [(x,y,z) | x <- [1..lim],
                           y <- [x..lim],
                           z <- [y..lim],
                           x*x+y*y == z*z]
```

SUPER-PROBLEMAS

Numeros Suaves (I)

- Si definimos **suaves** como la lista infinita ordena de los números suaves (aquellos cuyos únicos factores primos son 2,3 y 5), esa lista se puede definir como el 1 junto con la "fusion" de 3 listas: la lista de suaves multiplicados por 2, la lista de suaves multiplicada por 3 y la lista de suaves multiplicada por 5:

```
suaves :: [Int]
suaves = 1 : fusion3 (map (2*) suaves)
                     (map (3*) suaves)
                     (map (5*) suaves)
```

- Aquí **fusion3** recibe 3 listas ordenadas y devuelve una única lista con todos los elementos, ordenada y sin duplicados

Numeros Suaves (II)

- **fusion3** se puede definir en base a una función que fusione dos listas ordenadas:

```
fusion3 :: (Ord a) => [a] -> [a] -> [a] -> [a]
fusion3 xs ys zs = fusion xs (fusion ys zs)

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

- Nota: No es necesario considerar el caso de que alguna de las listas esté vacía ya que se trabaja con listas infinitas.

Triángulo de Pascal (I)

- Es sencillo crear una función recursiva para generar la siguiente fila:

```
sigFil :: [Integer] -> [Integer]
sigFil lis = 1:sigFilAux lis

sigFilAux :: [Integer] -> [Integer]
sigFilAux (x:y:res) = x+y : sigFilAux (y:res)
sigFilAux [x]       = [x]
sigFilAux []        = []
```

- Pero se puede hacer más compacto usando compresión de listas:

```
sigFil :: [Integer] -> [Integer]
sigFil lis = [x+y | (x,y) <- zip (0:lis) (lis++[0])]
```

Triángulo de Pascal (II)

- Si creamos una matriz infinita de filas del triángulo de Pascal:

```
pascal :: [[Integer]]
pascal = [1] : map sigFil pascal
```

- Entonces el programa principal sólo necesita ir coger las filas necesarias y aplicarles la función externa.
- La función externa recibía el número total de filas, el índice de fila actual y la fila, y la imprimia. Definimos su tipo de datos como un sinónimo:

```
type FunPas = Int -> Int -> [Integer] -> IO()
type FunPasN = Int -> [Integer] -> IO()
```

Triángulo de Pascal (III)

- El programa principal sería (usamos una función auxiliar y la notación **do** para mónadas IO):

```
problema2 :: FunPas -> Int -> IO()
problema2 f n = prob2Aux (f n) 0 (take n pascal)

prob2Aux :: FunPasN -> Int -> [[Integer]] -> IO()
prob2Aux fun i [] = return ()
prob2Aux fun i (fil:fils) =
   do
     fun i fil
     prob2Aux fun (i+1) fils
```

Triángulo de Pascal (IV)

- Un ejemplo de posible función procesadora de lineas sería (lineas centradas, escribe asteriscos si es impar)

```
proc1 :: FunPas
proc1 n i lis =
  putStrLn ((take (n-i) spcs) ++ (linea lis))
  where
    spcs = ' ':spcs
    linea l = foldl1 (++) (map valcar l)
    valcar x = if x `mod` 2 == 0
               then "  "
               else "* "
```

centrar linea

lista infinita de espacios

traducir números a espacios/asteriscos

Frecuenciador (I)

- El enfoque de la versión imperativa consistía en:
  1. **Recorrer los datos actualizando un diccionario de pares (dato, frecuencia)**
  2. **Ordenar la lista de pares (dato,frecuencia)**
- Este enfoque no casa bien en la versión funcional ya que deberíamos tener una estructura (el diccionario) que está siendo constantemente actualizada.
- Afortunadamente podemos lograr que la lista de pares (dato,frecuencia) se cree directamente:
  1. **Ordenamos los datos (los iguales quedarán agrupados)**
  2. **Los recorremos creando la lista de pares (dato,frec.)**
  3. **Ordenamos la lista de pares**

Frecuenciador (II)

- La función que crea la lista de pares partiendo de la lista ya ordenada sería:

```
type Cuenta a = (Int,a)

recuento :: (Eq a) => [a] -> [Cuenta a]
recuento [] = []
recuento (x:xs) = recaux xs (1,x)

-- Técnica del "parámetro acumulador"
recaux :: (Eq a) => [a] -> Cuenta a -> [Cuenta a]
recaux [] c = [c]
recaux (x:xs) (n,y) = if x == y
                      then recaux xs (n+1,y)
                      else (n,y) : recaux xs (1,x)
```

Frecuenciador (III)

- La función que ordena la lista es más problemática: Necesitamos un método de ordenación eficiente.
- Mirando en la literatura encontramos que el método más usado es la **ordenación rápida (QuickSort).**
- Es un método recursivo. En cada etapa estamos ordenando un trozo del vector, y:
  - Escogemos un valor cualquiera de ese trozo (el **pivote**)
  - Movemos los elementos menores a la parte izquierda, los mayores o iguales a la parte derecha y dejamos el pivote entre esos dos.
  - Ordenamos recursivamente la parte izquierda y la parte derecha.
  - Ese subvector ya está ordenado.

QuickSort en Pascal (I)


```
procedure OrdRap(var V: TVector);
var I,J : integer;
begin
  { Desordenar vector (Knuth shuffle algorithm) }
  for I := 0 to Lenght(V)-1 do
    V[I] ⇔ V[Random(Length(V)-I)]
  { Ordenación recursiva sobre todo el vector }
  OrdRapRec(V,0,Length(V)-1)
end;

procedure OrdRapRec(var V: TVector; Ini,Fin: integer);
{ Ordena V[Ini..Fin] }
var Fin_Men,Ini_May : integer;
begin
  if Ini < Fin then
  begin
    Particion(V,Ini,Fin,Fin_Men,Ini_May); { Redistribuir elems. }
    OrdRapRec(V,Ini,Fin_Men); { Ordena parte de menores }
    OrdRapRec(V,Ini_May,Fin); { Ordena parte de mayores }
  end { else caso_base }
end;
```

QuickSort en Pascal (II)


```
procedure Partición(var V: TVector; Ini,Fin: integer;
                    var Fin_Men, Ini_May: integer);
{ Reorganiza V[Ini..Fin] de manera que termine organizado en tres zonas:
  · V[Ini..Fin_Men] contiene elementos menores o iguales al pivote.
  · V[Fin_Men+1..Ini_May-1] contiene elementos iguales al pivote.
  · V[Ini_May..Fin] contiene elementos mayores o iguales al pivote.
  Ninguna zona se extiende en todo V[Ini..Fin] }
var
  Izda,Dcha : integer; Pivote : TClave;
begin
  Pivote := V[Ini].Clave; { Hay otras alternativas a elección de pivote }
  Izda := Ini; Dcha := Fin;
  while Izda <= Dcha do
  begin { Invariante: V[Ini..Izda-1] <= Pivote, V[Dcha+1..Fin] >= Pivote }
    while V[Izda].Clave < Pivote do Inc(Izda);
    while V[Dcha].Clave > Pivote do Dec(Dcha);
    if Izda <= Dcha then
    begin
      V[Izda] ⇔ V[Dcha];
      Inc(Izda); Dec(Dcha)
    end
  end;
  Fin_Men := Dcha; Ini_May := Izda
end;
```

QuickSort en Haskell

- Traduciendo el algoritmo a Haskell, tenemos:

```
qsort :: (Ord a) => [a] -> [a]
qsort [] = []
qsort (piv:xs) = qsort [x | x <- xs, x < piv]
                 ++ [piv] ++
                 qsort [x | x <- xs, x >= piv]
```

Frecuenciador (III)

- La función que dada la lista de datos devuelve la lista de pares frecuencia, dato (tipo Cuenta) sería:

```
programa :: (Ord a) => [a] -> [Cuenta a]
programa = qsort . recuento . qsort
```

- Si queremos escribir la lista de *n* cuentas formateada:

```
escribir :: (Show a) => Int -> [Cuenta a] -> IO()

escribir n [] = putStrLn ""
escribir n (c:cs)
  | n <= 0    = return ()
  | otherwise = do
                  putStrLn (trad c)
                  escribir (n-1) cs
  where
    trad (n,x) = (show n) ++ ": " ++ (show x)
```

Ejercicios (I)

- Generalizar la función **frases** de la transparencia 60 para que pueda trabajar con listas de cualquier tipo de datos, indicando la condición que permite detectar que un dato actua como separador de un grupo
- Por ejemplo, si tenemos una lista de enteros y los separadores son valores 0, un ejemplo de uso sería:

```
agrupador (\n -> n /= 0) [1,2,0,3,0,0,4,5,0,6] →
[[1,2],[3],[],[4,5],[6]]
```

- Transparencia 71: ¿Que hace **flip**? ¿Que función es **flip (:)**? ¿Como funciona realmente?

Ejercicios (II)

- ¿Que hace la siguiente función? Recuerde que el punto es la función composición $(f \cdot g)\ x = f\ (g(x))$. Puede ser conveniente averiguar primero su tipo de datos.

```
misterio = foldr ((.) . (:)) (\x -> x)
```

- Para hallar la raiz cuadrada de un valor, n, se puede usar la fórmula de Newton-Raphson.
  - Se parte de $x_0 = n/2$
  - Se itera la fórmula $x_{i+1} = (x_i + n / x_i)/2$
  - Hasta que $| x_{i+1} - x_i | < eps$
- Cree una función **`raiz n eps`** basada en este esquema