Estructuras de datos

Clases teóricas

por Pablo E. “Fidel” Martínez López

1. Presentación de la materia
   Tipos de datos algebraicos

Presentación de la materia

Equipo de trabajo

- Profesores
  - Pablo E. “Fidel” Martínez López
  - Cristian Sottile
  - Alejandro Castro
- Auxiliares y colaboradores
  - Horacio Valenzuela, Gonzalo Verón, Jonatan Maia, Federico Sandoval, Cristian Espíndola, Lucas Morón, Julián Bensal, Diego Moronha, Camila Scaglioni

Objetivos de la materia

- Definir distintos tipos de datos y operar con ellos
  - En particular, tipos para representar estructuras de datos (árboles, pilas, colas, etc.)
- Contrastar dos modelos de programación
  - Modelo denotacional (funcional)
  - Modelo destructivo (imperativo)
- Entender cuestiones de bajo nivel
  - En particular, manejo explícito de memoria

Contenidos

- **Unidad 1:** Tipos algebraicos
- **Unidad 2:** Tipos abstractos y eficiencia operacional
- **Unidad 3:** Modelo destructivo y estructuras de bajo nivel
- **Temas adicionales:** Hashing, Sorting, Grafos

Evaluación

- Dos parciales, con recuperatorio
  - Parcial 1: después de la Unidad 1
  - Parcial 2: después de la Unidad 2
  - Recuperatorios: después de la Unidad 3
- Evaluación integradora
  - Al final de la materia

Consejos para estudiar bien

- Resolver todos los ejercicios de las prácticas
  - Antes del examen
  - Siguiendo los conceptos dados en la materia
    (no sirve resolver de cualquier forma)
- No faltar a clase
- Dedicar horas de estudio adicional (al menos 8hs)
- Tener paciencia para madurar conceptos

Tipos de datos

Algunas definiciones básicas

- Dato
  - Unidad de información distinguible y manipulable
- Expresión
  - Combinación de símbolos que describe a un dato
  - Se forma a partir de constantes, variables y funciones
  - Pueden evaluarse para obtener su valor

Algunas definiciones básicas

- Tipo de datos
  - Atributo para clasificar expresiones
  - Impone restricciones a las combinaciones posibles
- Sistema de tipos
  - Conjunto de reglas para asignar tipo a las expresiones
  - Usualmente se ejecuta durante la compilación

Algunas definiciones básicas

- Función
  - Expresión que permite transformar otras expresiones
  - Transforma argumentos en un resultado
  - Se nombran mediante identificadores
  - Se escribe la función y a continuación sus argumentos
    - `multiplicar 2 3` da `6`
      (transforma al `2` y `3` en un `6`)
    - `sumar 1 (multiplicar 2 3)` da `7`
      (transforma al `1` y al `6` en un `7`)

Algunas definiciones básicas

- Estructura de datos
  - Forma particular de organizar datos para ser accedidos o manipulados de cierta manera (usualmente en forma eficiente)
  - Ejemplos:
    - El tablero de Gobstones
    - Listas
    - Registros

Modelos de computación

Modelo de computación

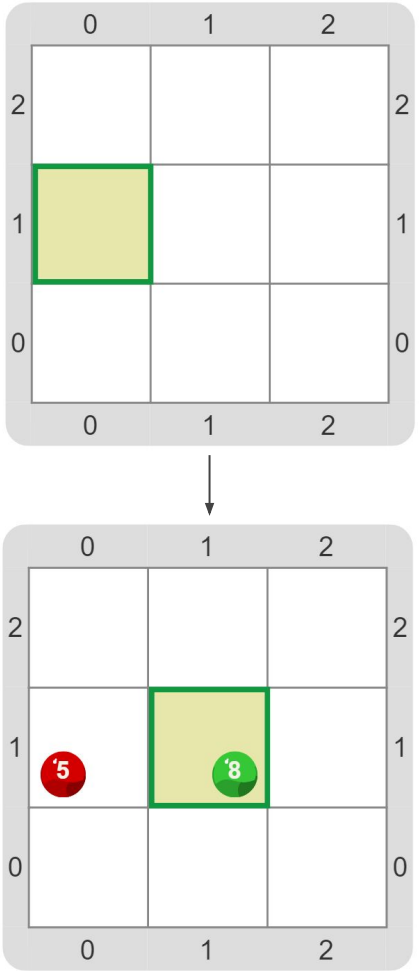
- Modelo destructivo
  - También llamado *imperativo*
  - Se ejecutan comandos que producen efectos
  - El programa va de un estado inicial a un estado final
- Modelo denotacional
  - También llamado *funcional*
  - Se evalúan expresiones que describen valores
  - El programa es una expresión y su resultado es el valor de esa expresión

Modelo de computación

- Modelo destructivo, ejemplo

```
program {
    Poner__Veces(Rojo, 5)
    Mover(Este)
    Poner__Veces(Verde, 8)
}

procedure Poner__Veces(cant,color) {
    repeat (cant) { Poner(color) }
}
```



Modelo de computación

- Modelo denotacional, ejemplo

```
program {
    return(sumar(dos(),sumar(dos(),3)))
}

function sumar(n,m) {
    return (n+m)
}

function dos() {
    return (2)
}
```

Modelo de computación

- En ambos modelos se transforma información
  - En el destructivo, se transforma el estado
  - En el denotacional, argumentos en un resultado
- Gobstones tiene ambos modelos
  - Los procedimientos transforman el estado del tablero
  - Las funciones transforman argumentos en un resultado
- Haskell es un lenguaje funcional
  - No posee comandos
  - Es más adecuado para estudiar estructuras de datos

Haskell

Haskell

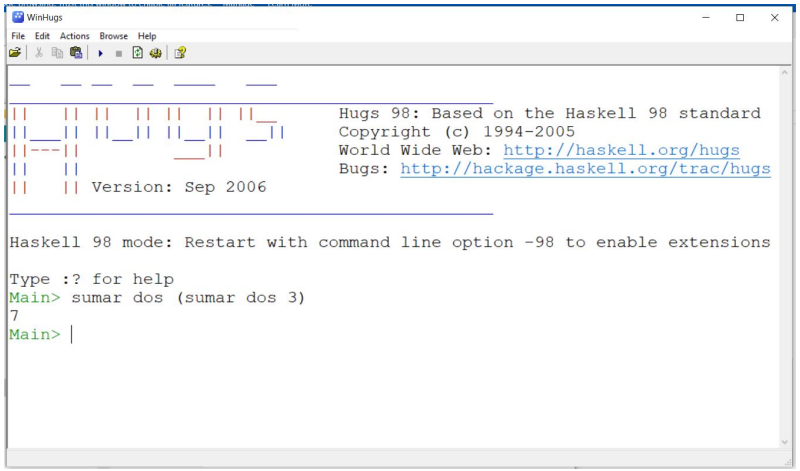
- Haskell es un lenguaje funcional puro
  - No hay memoria, ni estado de ningún tipo
  - Permite programar declarativamente
- Vamos a utilizar solamente un fragmento del mismo
  - NO vamos a aprender Haskell en profundidad
  - NO recomendamos leer de cualquier lado

Haskell

- Un programa Haskell es un conjunto de definiciones de funciones y constantes
- Se utiliza evaluando una expresión
- Ejemplo

```
sumar :: Int -> Int -> Int
sumar n m = n+m

dos :: Int
dos = 2
```



Haskell

- Características
  - Los parámetros van sin paréntesis ni comas
  - No hacen falta llaves ni **`return`** (se usa un **`=`**)
  - Cada definición puede indicar su tipo
    - El símbolo **`::`** se lee “tiene tipo”
    - Así, **`dos :: Int`** se lee “dos tiene tipo Int”

```
dos :: Int
dos = 2
```

Haskell

- Características
  - Los argumentos se ponen separados por espacios
  - Los paréntesis se usan diferente a otros lenguajes
    - Solamente para *agrupar* expresiones
  - Gobstones

```
doble(2)
mult(2,3)
doble(mult(2,3))
mult(doble(2),doble(2))
mult(mult(2,3),mult(2,3))
```

  - Haskell

```
doble 2
mult 2 3
doble (mult 2 3)
mult (doble 2) (doble 2)
mult (mult 2 3) (mult 2 3)
```

Haskell

- Características
  - Los comentarios de línea comienzan con `--`
  - Los comentarios de párrrafo van entre `{-` y `-}`
  - Ejemplos

```
{- Mostramos un ejemplo de función parcial.
   Recordemos que las funciones parciales pueden fallar -}

divisionEntera :: Int -> Int -> Int
-- PRECOND: m es distinto de 0
divisionEntera n m = div n m
```

Alternativa condicional en Haskell

- Construcciones de Haskell: alternativa condicional
  - **`if-then-else`**
    (similar al **`choose-when`** de Gobstones)

```
max :: Int -> Int -> Int
max n m = if (n>m)
            then n
            else m
```

    - ¡Las ramas son expresiones!
    - Siempre tiene que haber un **`else`**

Alternativa condicional en Haskell

- Construcciones de Haskell: alternativa condicional
  - Pueden anidarse varios **`if-then-else`**

```
signo :: Int -> Int
signo n = if (n==0)
            then 0
            else if (n>0)
                   then 1
                   else -1
```

```
signo :: Int -> Int
signo n = if      (n==0) then  0
          else if (n>0)  then  1
          else                -1
```

  - No hay multialternativa

Errores explícitos en Has

```
divisionEntera :: Int -> Int -> Int
-- PRECOND: m es distinto de 0
divisionEntera n m = div n m
```

- Construcciones de Haskell: error explícito
  - Para funciones parciales PUEDE usarse **`error`**

```
divisionEntera' :: Int -> Int -> Int
-- PRECOND: m es distinto de 0
divisionEntera' n m =
  if m==0
   then error "No se puede dividir por cero"
   else div n m
```

    - Es como el **`boom`** de Gobstones

Tipos básicos en Haskell

- Haskell tiene varios tipos básicos
  - Números: **`Int`**, **`Integer`**, **`Float`**, **`Double`** (**`1`**, **`2`**, **`3`**, …)
  - Caracteres: **`Char`** (**`'a'`**, **`'b'`**, **`'c'`**, …)
  - Booleanos: **`Bool`** (**`True`**, **`False`**)
  - Strings: **`String`** (**`"Hola"`**, **`"Mensaje"`**, **`""`**, …)
    - **`String`** es sinónimo de **`[Char]`**
    - **`"Hola"`** es equivalente a **`['H','o','l','a']`**
- Cada uno tiene su propia sintaxis y operaciones
  - Deben averiguar los detalles en el apunte

Tipos algebraicos

Tipos algebraicos: definición

- Además de los tipos básicos, pueden definirse otros
  - Incluyen variantes, registros y recursivos
- Se definen con la palabra clave **`data`**
- Se enumeran las formas de construir elementos
- Ejemplos:

```
data Dir     = Norte | Este | Sur | Oeste
data Persona = P String Int  String
                -- Nombre Edad DNI
```

Tipos algebraicos: enumerativos

- Tipos algebraicos variantes: enumerativos
  - Se dan los constructores que definen los casos
  - Cada constructor determina un elemento

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

  - Esta definición es similar a la de Gobstones

```
type Dir is variant {
  case Norte {}
  case Sur   {}
  case Este  {}
  case Oeste {}
}
```

Tipos algebraicos: registros

- Tipos algebraicos registro
  - Se da un único constructor con argumentos
  - El constructor tener nombre diferente al tipo

```
data Persona = P String Int String
               -- Nombre Edad DNI
```

  - Esta definición es similar a la de Gobstones

```
type Persona is record {
  field nombre // String
  field edad   // Número
  field dni    // String
}
```

Tipos algebraicos: ejemplos

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

data Persona =
       P String Int  String
      -- Nombre Edad DNI
```

- Tipos algebraicos
  - Ejemplos de uso

```
direccionDefault :: Dir
direccionDefault = Este

yo, juan, andrea :: Persona
yo = P "Fidel" 53 "20XXXXXXX"
juan = P "Juan" 23 "23345322"
andrea = P "Andrea" 42 "33834673"
```

```
function yo() {
 return (
  Persona(
    nombre <- "Fidel"
  , edad   <- 53
  , dni    <- "20XXXXXXX"
 ))
}
```

```
type Persona is record {
  field nombre // String
  field edad   // Número
  field dni    // String
}
```

  - Atención al constructor diferente

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

Tipos algebraicos: *pattern matching*

- Acceso a tipos algebraicos: *pattern matching*
  - Los constructores se pueden usar para acceder
  - Para eso se usan en los parámetros

```
siguienteDir :: Dir -> Dir
siguienteDir Norte = Este
siguienteDir Este  = Sur
siguienteDir Sur   = Oeste
siguienteDir Oeste = Norte
```

```
function siguienteDir(d) {
  return(matching (d) select
           Este  on Norte
           Sur   on Este
           Oeste on Sur
           Norte on Oeste
           boom("") otherwise)
}
```

  - Es similar a la alternativa indexada de Gobstones

Acceden al dato

Construyen datos

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

Tipos algebraicos: *pattern matching*

- Acceso a tipos algebraicos: *pattern matching*
  - También hay una alternativa indexada como expresión

```
siguienteDir' :: Dir -> Dir
siguienteDir' d =
   case d of
    Norte -> Este
    Este  -> Sur
    Sur   -> Oeste
    Oeste -> Norte
```

```
function siguienteDir(d) {
  return(matching (d) select
           Este  on Norte
           Sur   on Este
           Oeste on Sur
           Norte on Oeste
           boom("") otherwise)
}
```

  - Es similar a la alternativa indexada de Gobstones

Acceden al dato

Construyen datos

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

Tipos algebraicos: *pattern matching*

- Acceso a tipos algebraicos: *pattern matching*
  - Los constructores se pueden usar para acceder
  - Para eso se usan en los parámetros
    - Si el parámetro coincide, usa dicha ecuación
    - Si no, continúa verificando la siguiente ecuación
  - Solo se aplica a constructores de tipos algebraicos

```
data Persona =
       P String Int  String
      -- Nombre Edad DNI
```

Tipos algebraicos: *pattern mat*

- Acceso a tipos algebraicos: *pattern matching*
  - Los constructores se pueden usar para acceder
  - También se puede usar para definir observadores

```
nombre :: Persona -> String
nombre (P n e d) = n

edad :: Persona -> Int
edad (P n e d) = e
```

  - Observar el uso de variables como argumento de **P**
  - En Gobstones los observadores están predefinidos

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

Tipos algebraicos: *pattern matching*

- Acceso a tipos algebraicos: *pattern matching*
  - El orden de las ecuaciones importa
  - Comparar

```
esEste' :: Dir -> Bool
esEste' Este  = True
esEste' Norte = False
esEste' Sur   = False
esEste' Oeste = False
```

```
esEste :: Dir -> Bool
esEste Este = True
esEste d    = False
```

  - ¿Son equivalentes?

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

Tipos algebraicos: *pattern matching*

- Acceso a tipos algebraicos: *pattern matching*
  - El orden de las ecuaciones importa
  - Comparar

```
esEsteMal :: Dir -> Bool
esEsteMal d    = False
esEsteMal Este = True
```

```
esEste :: Dir -> Bool
esEste Este = True
esEste d    = False
```

  - ¿Son equivalentes?
  - ¿Cuál es la diferencia?

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

Tipos algebraicos: *pattern matching*

- Acceso a tipos algebraicos: *pattern matching*
  - El orden de las ecuaciones importa
  - Si el parámetro no se usa, se puede usar un comodín

```
esEste'' :: Dir -> Bool
esEste'' Este = True
esEste'' _    = False
```

```
esEste :: Dir -> Bool
esEste Este = True
esEste d    = False
```

  - Son equivalentes
  - Puedo ahorrarme pensar un nombre para el parámetros

Tipos algebraicos: *pattern matching*

- Acceso a tipos algebraicos: *pattern matching*
  - El orden de las ecuaciones importa

```
disyuncionPM :: Bool -> Bool -> Bool
disyuncionPM True  True  = True
disyuncionPM True  False = True
disyuncionPM False True  = True
disyuncionPM False False = False
```

    - ¿Cómo hacerlo con menos ecuaciones?

Tipos algebraicos: *pattern matching*

- Acceso a tipos algebraicos: *pattern matching*
  - El orden de las ecuaciones importa

```
disyuncionPM :: Bool -> Bool -> Bool
disyuncionPM False False = False
disyuncionPM _     _     = True
```

  - ¿Cómo hacerlo con menos ecuaciones?
    - Aprovechando el orden

Tipos algebraicos: *pattern mat*

```
data Persona =
       P String Int  String
      -- Nombre Edad DNI
edad :: Persona -> Int
edad (P n e d) = e
```

- Acceso a tipos algebraicos: *pattern matching*
  - Pattern matching vs. funciones observadoras

```
esMayorDeEdad' :: Persona -> Bool
esMayorDeEdad' (P _ e _) = e >= 18

esMayorDeEdad :: Persona -> Bool
esMayorDeEdad p = edad p >= 18
```

    - ¿Qué ventajas y desventajas tiene cada una?

Tipos algebraicos: sumas

- Tipos algebraicos variantes: sumas
  - Los constructores pueden tener argumentos

```
data Gusto = Chocolate | Sambayon | DDL | Frutilla
data Helado = Vasito Gusto | Cucurucho Gusto Gusto
            | Pote Gusto Gusto Gusto
```

    - Los tipos en posición de argumento van SOLAMENTE en las declaraciones **`data`**
    - Significa que, por ejemplo, el constructor **`Vasito`** seguido de un **`Gusto`** es un valor válido de tipo **`Helado`**

Tipos algebraicos: suma

```
data Gusto = Chocolate | Sambayon
           | DDL | Frutilla
data Helado = Vasito Gusto
            | Cucurucho Gusto Gusto
            | Pote Gusto Gusto Gusto
```

- Tipos algebraicos variantes: sumas
  - Los constructores pueden tener argumentos

```
miHeladoFavorito :: Helado
miHeladoFavorito = Cucurucho Chocolate Sambayon

unVasito :: Helado
unVasito = Vasito DDL
```

    - Observar que después de **`Vasito`** NO va el tipo **`Gusto`**, sino UN VALOR de ese tipo (en este caso, **`DDL`**)

Tipos algebraicos: suma

```
data Gusto = Chocolate | Sambayon
           | DDL | Frutilla
data Helado = Vasito Gusto
            | Cucurucho Gusto Gusto
            | Pote Gusto Gusto Gusto
```

- Tipos algebraicos variantes: sumas
  - En el *pattern matching*, se usan variables

```
esHeladoSerio :: Helado -> Bool
esHeladoSerio (Vasito g)         = esGustoSerio g
esHeladoSerio (Cucurucho g1 g2) = esGustoSerio g1
                                 && esGustoSerio g2
esHeladoSerio (Pote g1 g2 g3)    = esGustoSerio g1
                                 && esGustoSerio g2
                                 && esGustoSerio g3
```

    - Observar que DEBEN ser variables diferentes
    - ¿Cómo se define **`esGustoSerio`**?

Tipos algebraicos: suma

```
data Gusto = Chocolate | Sambayon
           | DDL | Frutilla
data Helado = Vasito Gusto
            | Cucurucho Gusto Gusto
            | Pote Gusto Gusto Gusto
```

- Tipos algebraicos variantes: sumas
  - En el *pattern matching*, se usan variables

```
sinFrutilla :: Helado -> Helado
sinFrutilla (Vasito g)         = Vasito (cambiarFrutilla g)
sinFrutilla (Cucurucho g1 g2) = Cucurucho
                                  (cambiarFrutilla g1)
                                  (cambiarFrutilla g2)
sinFrutilla (Pote g1 g2 g3)    = Pote    (cambiarFrutilla g1)
                                  (cambiarFrutilla g2)
                                  (cambiarFrutilla g3)
```

    - ¿Cómo se define **`cambiarFrutilla`**?

Tuplas y tipos polimórficos

Tuplas

- Las tuplas son estructuras de datos predefinidas
  - Son similares a un registro, pero sin nombres de campo
  - Se escriben entre paréntesis y con comas
    (tanto los valores como los tipos)

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

unaTerna :: (Int, String, Bool)
unaTerna      = (3, "", True)

unParDePares :: ((Int, Bool), (String, Int))
unParDePares = ((2, True), ("Hola", 17))
```

    - Son como el producto cartesiano en matemáticas

Tuplas

- Las tuplas son estructuras de datos predefinidas
  - Los paréntesis y la coma funcionan como un *constructor*
  - Se puede hacer *pattern matching* sobre tuplas

```
fst :: (Int, Bool) -> Int
fst (n,b) = n

snd :: (Int, Bool) -> Bool
snd (n, b) = b
```

  - ¿Es necesario que los elementos sean de tipos fijos?

Tipos polimórficos

- ¿Y si el elemento no está restringido?
  - Se utilizan *variables de tipo*
  - En diferentes usos, puede tomar diferentes tipos

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

```
> fst (2, True)
2

> fst ("Hola", 17)
"Hola"
```

  - ¡Observar que es *la misma función* **`fst`**!

Tipos polimórficos

- Polimorfismo paramétrico
  - Permite definir funciones genéricas
    - Una sola definición opera sobre muchos tipos
    - No es necesario redefinir lo mismo una y otra vez
  - Permite definir estructuras de datos genéricas
    - Estructuras como “contenedores de datos”
    - Independencia del contenedor respecto del tipo de dato

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

Tipos polimórficos

- Polimorfismo paramétrico
  - La instaciación la hace el sistema de tipos
  - Haskell se encarga de reemplazar esas variables

```
> fst (2, True)
2
```

```
> fst ("Hola", 17)
"Hola"
```

  - En el primer caso, **`fst :: (Int,Bool)->Int`** (con **`a<-Int`**, **`b<-Bool`**)
  - En el segundo caso, **`fst :: (String,Int)->String`** (con **`a<-String`**, **`b<-Int`**)

Tipos polimórficos

- Polimorfismo paramétrico
  - Ejemplos

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

siempreSiete :: a -> Int
siempreSiete x = 7

const :: a -> b -> a
const x y = x
```

    - Observar que en cada caso, no se opera con el parámetro

Tipos polimórficos

- Polimorfismo paramétrico
  - Si a la función le importa la estructura, pero no los datos específcos, puede ser *polimórfica*
    - Es posible mientras no se opere con el parámetro
    - Si se usa alguna operación no polimórfica, no es posible
  - Si importa el tipo del dato, la función es *monomórfica*

```
succ :: Int -> Int   -- ¿succ :: a -> a? NO
succ n = n+1
```

Listas

Listas

- Las listas son estructuras de datos predefinidas
  - Son similares a las listas de Gobstones
  - Pero hay algunas diferencias
    - La notación `[1,2,3]` es abreviatura de `1:2:3:[]`
    - Las operaciones de `(:)` y `[]` son constructores
    - La operación de agregar `(++)` no es predefinida
    - Así, son equivalentes

```
[1,2,3]        1:2:3:[]              1:[2,3]

[1]++[2,3]     (1:[])++(2:[3])
```

Listas

- Las listas son estructuras de datos predefinidas
  - `(:) :: a -> [a] -> [a]`
    - Se lee ***cons***
    - Toma un elemento de algún tipo y una lista del mismo tipo, y describe la lista que resulta de agregar el elemento adelante
  - `[] :: [a]`
    - Se lee ***lista vacía*** (o ***nil***)
    - Es una lista de algún tipo a instanciar

Listas

- Los constructores pueden usarse para *pattern matching*

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

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

```
null :: [a] -> Bool
null [] = True
null _  = False

noEsVacia :: [a] -> Bool
noEsVacia (_:_) = True
noEsVacia _     = False
```

  - Las variables de lista se suelen terminar con la letra s
  - Permite indicar que son muchos elementos de ese tipo

Listas

- Los constructores pueden usarse para *pattern matching*
- Recordar: **`head`** y **`tail`** son operaciones parciales
  - No están definidas para la lista vacía
  - Podrían definirse indicando explícitamente el error

```
head :: [a] -> a
head (x:_) = x
head _     = error "head no se puede usar con []"
```

Listas

- Los constructores pueden usarse para *pattern matching*
- Pero la función de agregar **`(++)`** NO es constructora
  - Significa que NO se puede hacer *patter matching* con ella
  - O sea, NO se puede hacer esto:

```
f (xs++ys) = …
```

    - Es un error de construcción

Listas

- Los constructores pueden usarse para *pattern matching*
- Más ejemplos de *pattern matching* válidos sobre listas:

```
esSingular :: [a] -> Bool
esSingular (_:[]) = True
esSingular _      = False

segundo :: [a] -> a
segundo (_:x:_) = x
segundo _       = error "No hay 2 elementos"
```

Listas

- ¿Cómo hacer recorridos con listas?
  - Será el tema de la próxima clase

Resumen

Resumen

- Modelo de computación denotacional
- Tipos de datos
  - Tipos básicos
  - Tipos algebraicos
    - *Pattern matching*
  - Tuplas, listas
- Tipos polimórficos