Tema 6. Tipos algebraicos

data <tipo> = <C_1> <tipo(s)> | <C_2> <tipo(s)> | .....

- *Se definen mediante sus constructoras* C_1 , C_2 , ...
- *Pueden ser polimórficos y recursivos*

Ejs: **data** Dia = Lu | Ma | Mi | Ju | Vi | Sa | Do

data Bool = False | True

- *Funciones definidas mediante ajuste de patrones*

festivo :: Dia –> Bool
festivo Do = True
festivo d = False

not :: Bool –> Bool
not True = False
not False = True

Enumeraciones

Ej: **data** Dia = Lu | Ma | Mi | Ju | Vi | Sa | Do

Para definir Dia *como tipo enumerado:*

a) Lo declaramos instancia de Enum:

instance Enum Dia **where**

fromEnum Lu = 0

\- - - - - - - - - - - - - - - - - -

fromEnum Do = 6

toEnum 0 = Lu

\- - - - - - - - - - - - - - - - - -

toEnum 6 = Do

b) o derivamos la instancia automáticamente("deriving")

Clausula “deriving” (1)

```
data Dia =  Lu | Ma | Mi | Ju | Vi | Sa | Do
            deriving (Eq, Ord, Enum, Show)
```

Por ser instancia de Eq, Ord, Enum:

? Lu == Ma	? Ju < Sa	? pred Sa == succ Mi
False	True	False

Por ser instancia de Enum, Show:

? succ Vi	? pred Lu	? enumFromTo Lu Do
Sa	Error	[Lu,Ma,Mi,Ju,Vi,Sa,Do]

Clausula “deriving” (2)

```
data Dia =  Lu | Ma | Mi | Ju | Vi | Sa | Do
                 deriving (Eq, Ord, Enum, Show)
```

Funciones que usan operaciones heredadas:

```
festivo, laborable :: Dia –> Bool
festivo d = (d==Do)
laborable d = (d >= Lu) && (d <=Vi)

siguiente, anterior :: Dia –> Dia
siguiente d = if (d==Do) then Lu else succ d
anterior d = if (d==Lu) then Do else pred d
```

Tipos recursivos

```
data Nat =  Cero | Suc Nat
                    deriving  (Eq, Ord, Show)
```

Constructores: Cero :: Nat *y* Suc :: Nat –> Nat

Funciones definidas sobre Nat:

suma :: Nat –> Nat –> Nat	aInt :: Nat –> Int
suma Cero y = y	aInt Cero = 0
suma (Suc x) y = Suc(suma x y)	aInt (Suc x) = (aInt x) +1

? suma (Suc (Suc Cero)) (Suc Cero)	? aInt (Suc (Suc Cero))
Suc (Suc (Suc Cero))	2

Ejemplo: instancia de la clase Eq

```
infix :/
data MiRacional = Integer :/ Integer
```

- *Si* "**deriving** Eq" *entonces la igualdad es la estructural:*

  ? (4:/5) == (12:/15)
  False

- *Declarando la instancia con la siguiente definición de* (==):

```
instance Eq MiRacional where
(x:/y) == (x':/y')  =  (x*y'== y*x')
```

se obtiene como resultado:

? (4:/5) == (12:/15)
True

Ejemplo: instancia de la clase Show

data MiRacional = Integer :/ Integer

- *Si añadimos* “**deriving** Show”: ? 14:/21

  14:/21

- *Declarando la instancia con la siguiente definición de* **show**

```
instance Show MiRacional where
   show (x:/y) =  show (div x z) ++ “:/”++ show (div y z)
                  where z = mcd x y
```

se obtiene como resultado:

? 14:/21	? 14:/7
2:/3	2:/1

Ejercicio: mcd (máximo común divisor)

Tipo polimórfico predefinido “Maybe”

data Maybe α = Nothing | Just α
deriving (Eq, Ord, Read, Show)

Constructores: Nothing :: Maybe α
Just :: α –> Maybe α

Instancia (polimórfica) de las clases Eq, Ord, Show, Read

instance Eq α => **Eq** (Maybe α) .....

Ej:

? Just 3 == Just 5	? Nothing < Just 0	? Just 2
False	True	Just 2

Uso del tipo Maybe

Funciones predefinidas sobre Maybe :

maybe :: β –> (α –> β) –> Maybe α –> β

maybe n f Nothing = n

maybe n f (Just x) = f x

lookup :: Eq α => α –> [(α,β)] –> Maybe β

lookup k []   = Nothing

lookup k ((x,y): res)

    | k==x   = Just y

    | otherwise   = lookup k res

Tipo polimórfico predefinido Either

data Either α β = Left α | Right β
deriving (Eq, Ord, Read, Show)

Constructores: Left :: α –> Either α β
Right :: β –> Either α β

Instancia (polimórfica) de las clases Eq, Ord, Show, Read

instance (Eq α, Eq β) => **Eq** (Either α β) .....

Ej: c, b:: Either Char Int c = Left 'j' b = Left 'x'

? c == b	? c < b	? c < Right 1
False	True	True

Uso del tipo Either

*Función **pre**definida sobre* Either :

either :: (α –> γ) –> (β –> γ) –> Either α β –> γ

either f g (Left x) = f x

either f g (Right y) = g y

Función definida usando la anterior

ord_long:: Either Char String –> Int

ord_long = either ord length

? ord_long (Left ‘a’)
97

? ord_long (Right “ola marina”)
10

Tipos polimórficos y recursivos

$$\textbf{data}\ \text{Lista}\ \alpha\ =\ \text{Vac}\ |\ \text{Cons}\ \alpha\ (\text{Lista}\ \alpha)$$
$$\textbf{deriving}\ (\text{Eq, Ord, Show})$$

Constructores:

Vac :: Lista α — Vac $\approx$ []

Cons :: α –> Lista α –> Lista α — Cons $\approx$ (:)

Funciones polimórficas definidas sobre Lista α:

longitud :: Lista α –> Int

longitud Vac = 0

longitud (Cons x s) = 1+ longitud s

Arboles binarios

data Arbin α = Hoja α | Unir (Arbin α) (Arbin α)

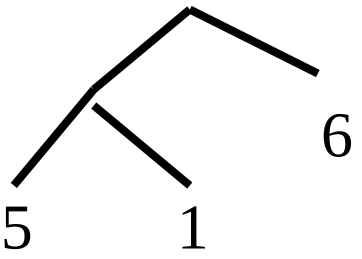
a1:: Arbin Int   a2:: Arbin Char

a1 = Unir (Unir (Hoja 5) (Hoja 1)) (Hoja 6)

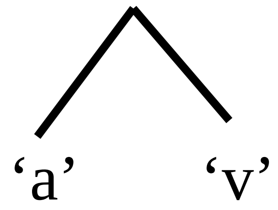
a2 = Unir (Hoja ‘a’) (Hoja ‘v’)

representan los árboles:

a1 =



a2 =



Funciones sencillas sobre Arbin α

prof :: Arbin α –> Int *(profundidad)*

prof (Hoja x) = 0

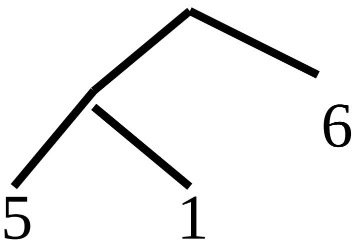
prof (Unir ai ad) = 1 + max (prof ai) (prof ad)

tamaño :: Arbin α –> Int *(número de hojas)*

tamaño (Hoja x) = 1

tamaño (Unir ai ad) = tamaño ai + tamaño ad

a1 =



? tamaño a1 3

? prof a1 2

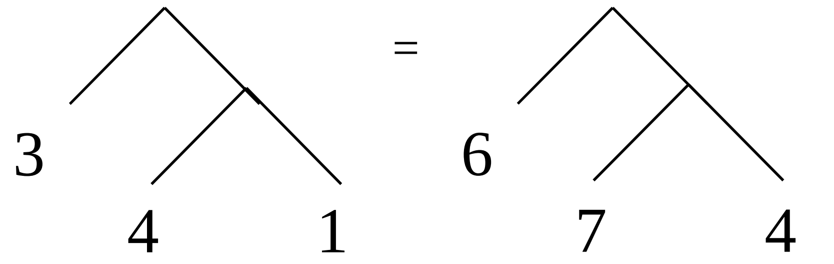
Función tipo “map” sobre Arbin α

maparbin :: (α–>β) –> Arbin α –> Arbin β

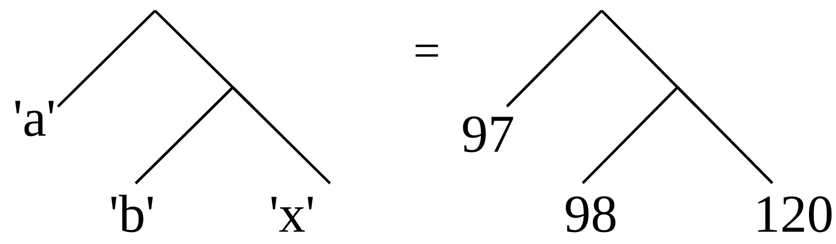
maparbin f (Hoja x) = Hoja (f x)

maparbin f (Unir ai ad) = Unir (maparbin f ai)(maparbin f ad)

maparbin (+3)



maparbin ord



Función tipo “fold” sobre Arbin α

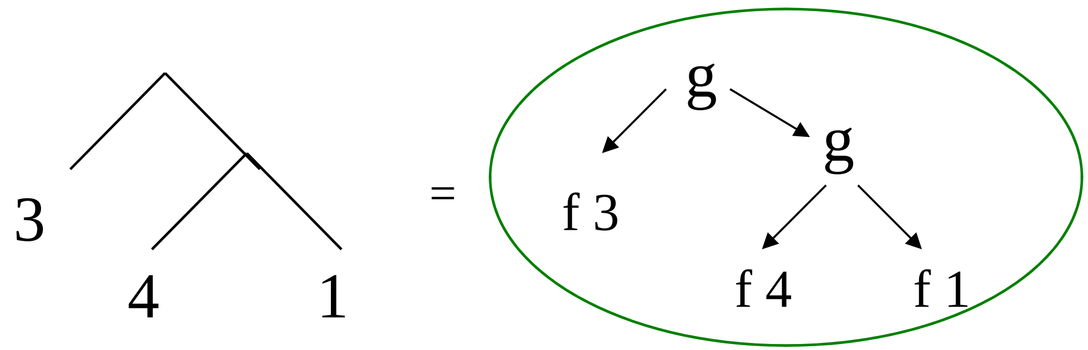
foldarbin :: (α–>β) –> (β–>β–>β) –> Arbin α –>β

foldarbin f g (Hoja x) = f x

foldarbin f g (Unir ai ad) = g (foldarbin f g ai) (foldarbin f g ad)

Idea:

foldarbin f g



= g (f 3) (g (f 4) (f 1))

tamaño = foldarbin (const 1) (+)

prof = foldarbin (const 0) g **where** g m n = 1+ max m n

Arbin α instancia de Eq y Show

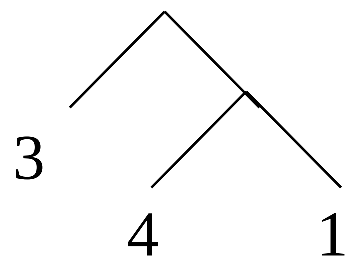
data Arbin α = Hoja α | Unir (Arbin α) (Arbin α)

deriving Eq

"La igualdad estructural es adecuada"

PERO si deseamos mostrar en pantalla los árboles de forma:

mostrar



=

```
        1
    *
        4
*
    3
```

instance Show α => Show (Arbin α)
  where show = mostrar

en vez de **deriving** Show

Ejercicio: *definir la función* mostrar :: Show α => Arbin α –> String

Demostración por inducción en Arbin α

Probar formalmente el "teorema" siguiente:

$$\forall ar{:}Arbin\ \alpha, \qquad prof\ ar < tamaño\ ar \leq 2^{\,prof\ ar}$$

Demostración:

- caso simple (ar = Hoja x)

  ¿ $prof\ (Hoja\ x) < tamaño\ (Hoja\ x) \leq 2^{\,prof\ (Hoja\ x)}$?

- hipótesis de inducción: cierto el teorema para ai y para ad

- caso general (ar = Unir ai ad)

  ¿ $prof\ (Unir\ ai\ ad) < tamaño\ (Unir\ ai\ ad) \leq 2^{\,prof\ (Unir\ ai\ ad)}$?

  (usando la h.i. sobre ai y ad)

Inducción estructural

- *Dado un tipo algebraico,* ***demostrar una propiedad*** *sobre dicho tipo es probarla para cada uno de sus casos (con cada constructor)*

- *Si el tipo algebraico es recursivo, dicha demostración es por* ***inducción estructural*** *que consiste en*
  - casos simples: *probar la propiedad directamente para cada caso no recursivo*
  - hipótesis de inducción: *suponer la propiedad demostrada para las instancias internas recursivas*
  - caso general: *probarla para los casos recursivos (usando la h.i.)*

Arboles binarios de búsqueda

data Arbus α = Vac | Nod (Arbus α) α (Arbus α)

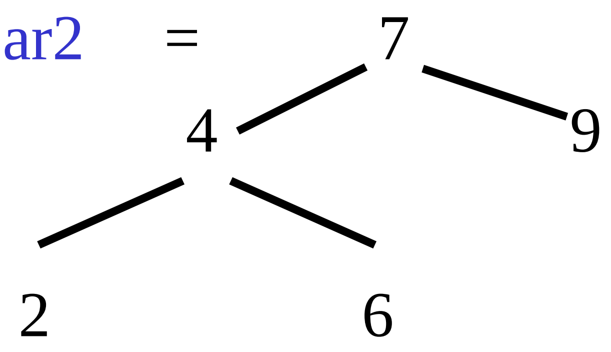
ar1, ar2:: Arbus Int   ar3:: Arbus String

ar1 = Nod (Nod Vac 2 Vac) 4 (Nod Vac 6 Vac)

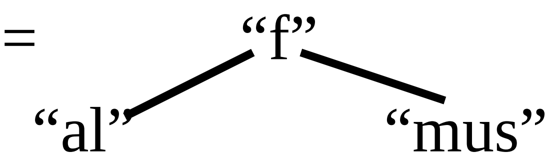
ar2 = Nod ar1 7 (Nod Vac 9 Vac)

ar3 = Nod (Nod Vac “al” Vac) “f” (Nod Vac “mus” Vac)

ar2 =



ar3 =



Funciones sobre Arbus α (1)

aplanar :: Arbus α –> [α]

aplanar Vac = []

aplanar (Nod ai r ad) = aplanar ai ++ [r] ++ aplanar ad

estaOrd :: Ord α =>Arbus α –> Bool

estaOrd = ordenada • aplanar

? aplanar ar2	? estaOrd ar2
[2,4,6,7,9]	True

Ejercicio: definir la función ordenada :: Ord α => [α] –> Bool

Funciones sobre Arbus α (2)

```
esta :: Ord α => α –> Arbus α –> Bool
esta x Vac = False
esta x (Nod ai r ad)    | x<r = esta x ai
                        | x==r = True
                        | x>r = esta x ad

meter :: Ord α => α –> Arbus α –> Arbus α
meter x Vac = Nod Vac x Vac
meter x (Nod ai r ad)  | x<r  = Nod (meter x ai) r ad
                       | x==r = Nod ai r ad
                       | x>r   = Nod ai r (meter x ad)
```

Funciones sobre Arbus α (3)

borrar :: Ord α => α –> Arbus α –> Arbus α

borrar x Vac = Vac

borrar x (Nod ai r ad) | x<r = Nod (borrar x ai) r ad
| x==r = une ai ad
| x>r = Nod ai r (borrar x ad)

donde la función une *debe cumplir la propiedad:*

aplanar (une ai ad) = aplanar ai ++ aplanar ad

(para ai, ad :: Arbus α *y con todos los nodos de* ai *menores que todos los de* ad*)*

Ejercicio: dar definiciones alternativas para la función une.