"Records"

Sendo p um valor do tipo PontoC, **p {xx=0}** é um novo valor com o campo xx=0 e os restantes campos com o valor que tinham em p.

Exemplos:

```
p1 {cor = Amarelo}  ≡  Pt {xx=3.2, yy=5.5, cor=Amarelo)
p3 {xx=0, yy=0}  ≡  Pt {xx=0, yy=0, cor=Verde}
```

```
simetrico :: PontoC -> PontoC
simetrico p = p {xx=(yy p), yy=(xx p)}
```

É possível ter campos etiquetados em tipos com mais de um construtor.
Um campo não pode aparecer em mais do que um tipo, mas dentro de um tipo pode aparecer associado a mais de um construtor, desde que tenha o mesmo tipo.

Exemplo:

```
data EX = C1 { s :: Int, r :: Float }
        | C2 { s :: Int, w :: String }
```


Polimorfismo paramétrico

Com já vimos, o sistema de tipos do Haskell incorpora tipos polimórficos, isto é, tipos com variáveis (*quantificadas universalmente*, de forma implícita).

Exemplos:

Para qualquer tipo a, [a] é o tipo das listas com elementos do tipo a.

Para qualquer tipo a, (ArvBin a) é o tipo das árvores binárias com nodos do tipo a.

As variáveis de tipo podem ser vistas como *parâmetros (dos constructores de tipos)* que podem ser substituídos por tipos concretos. Esta forma de polimorfismo tem o nome de ***polimorfismo paramétrico***.

Exemplo:

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

```
length [5.6,7.1,2.0,3.8]      ⇒  4
length ['a','b','c']          ⇒  3
length [(3,True),(7,False)]   ⇒  2
```

```
Prelude> :t length
length :: forall a. [a] -> Int
```

O tipo **[a]->Int** não é mais do que uma abreviatura de $\forall$***a.[a]->Int*** :

*"**para todo o tipo** a, [a]->Int é o tipo das funções com domínio em [a] e contradomínio Int".*

Polimorfismo *ad hoc* (sobrecarga)

O Haskell incorpora ainda uma outra forma de polimorfismo que é a ***sobrecarga** de funções*. Um mesmo identificador de função pode ser usado para designar funções computacionalmente distintas. A esta característa também se chama ***polimorfismo ad hoc***.

Exemplos:

O operador (+) tem sido usado para somar, tanto valores inteiros como valores decimais.

O operador (==) pode ser usado para comparar inteiros, caracteres, listas de inteiros, strings, booleanos, ...

Afinal, qual é o tipo de (+) ? E de (==) ?

A sugestão

```
(+)  :: a -> a -> a
(==) :: a -> a -> Bool
```

não serve, pois são tipos demasiado genéricos !

Faria com que fossem aceites espressões como, por exemplo:

('a' + 'b') , (True + False) , ("esta'" + "errado") ou (div == mod) ,

e estas expressões resultariam em **erro**, pois estas operações não estão preparadas para trabalhar com valores destes tipos.

Em Haskell esta situação é resolvidas através de **tipos qualificados** *(qualified types)*, fazendo uso da noção de **classe**.

Tipos qualificados

Conceptualmente, um tipo qualificado pode ser visto como um tipo polimórfico só que, em vez da quantificação universal da forma *"para todo o tipo a, ..."* vai-se poder dizer *"para todo o tipo a que pertence à classe C, ..."* . Uma classe pode ser vista como um conjunto de tipos.

Exemplo:

Sendo **Num** uma classe *(a classe dos números)* que tem como elementos os tipos: Int, Integer, Float, Double, ..., pode-se dar a (+) o tipo preciso de:

$\forall$***a***$\in$***Num.a->a->a***

o que em Haskell se vai escrever: `(+) :: Num a => a -> a -> a`

e lê-se: *"para todo o tipo a que pertence à classe Num, (+) tem tipo a->a->a".*

Uma classe surge assim como uma forma de classificar tipos (quanto às funcionalidades que lhe estão associadas). Neste sentido as classes podem ser vistas como os *tipos dos tipos*.

Os tipos que pertencem a uma classe também serão chamados de ***instâncias*** da classe.

A capacidade de *qualificar* tipos polimórficos é uma característica inovadora do Haskell.

Classes & Instâncias

Uma ***classe*** estabelece um conjunto de assinaturas de funções (os *métodos da classe*). Os tipos que são declarados como ***instâncias*** dessa classe têm que ter definidas essas funções.

Exemplo: A seguinte declaração (simplificada) da classe `Num`

```
class Num a where
    (+) :: a -> a -> a
    (*) :: a -> a -> a
```

impõe que todo o tipo `a` da classe `Num` tenha que ter as operações `(+)` e `(*)` definidas.

Para declarar `Int` e `Float` como elementos da classe `Num`, tem que se fazer as seguintes *declarações de instância*

```
instance Num Int where
    (+) = primPlusInt
    (*) = primMulInt
```

```
instance Num Float where
    (+) = primPlusFloat
    (*) = primMulFloat
```

Neste caso as funções *primPlusInt, primMulInt, primPlusFloat e primMulFloat* são funções primitivas da linguagem.

Se `x::Int` **e** `y::Int` **então** `x + y` ≡ ``x `primPlusInt` y``
Se `x::Float` **e** `y::Float` **então** `x + y` ≡ ``x `primPlusFloat` y``

Tipo principal

O **tipo principal** de uma expressão ou de uma função é o tipo mais geral que lhe é possível associar, de forma a que todas as possíveis instâncias desse tipo constituam ainda tipos válidos para a expressão ou função.

Qualquer expressão ou função válida tem um tipo principal *único*. O Haskell *infere* sempre o tipo principal das expressões ou funções, mas é sempre possível associar tipos mais específicos (que são instância do tipo principal).

Exemplo: O tipo principal inferido pelo haskell para o operador (+) é

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

Mas,

```
(+) :: Int -> Int -> Int
(+) :: Float -> Float -> Float
```

são também tipos válidos dado que tanto `Int` como `Float` são instâncias da classe `Num`, e portando podem substituir a variável a.

Note que `Num a` não é um tipo, mas antes uma restrição sobre um tipo. Diz-se que `(Num a)` é o ***contexto*** para o tipo apresentado.

Exemplo:

```
sum [] = 0
sum (x:xs) = x + sum xs
```

O tipo principal da função sum é

```
sum :: Num a => [a] -> a
```

- `sum::[a]->a` seria um tipo demasiado geral. Porquê ?
- Qual será o tipo principal da função `product` ?

Definições por defeito

Relembre a definição da função pré-definida `elem`:

```
elem x [] = False
elem x (y:ys) = (x==y) || elem x ys
```

Qual será o seu tipo ?

É necessário que (==) esteja definido para o tipo dos elementos da lista.

Existe pré-definida a classe **`Eq`**, dos tipos para os quais existe uma operação de igualdade.

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    -- Minimal complete definition: (==) or (/=)
    x == y = not (x /= y)
    x /= y = not (x == y)
```

Esta classe establece as funções `(==)` e `(/=)` e, para além disso, fornece também **definições por defeito** para estes métodos *(default methods)*.

Caso a definição de uma função seja omitida numa declaração de instância, o sistema assume a definição por defeito feita na classe. Se existir uma nova definição do método na declaração de instância, será essa definição a ser usada.

Exemplos de instâncias de Eq

O tipo `Cor` é uma instância da classe Eq com `(==)` definido como se segue:

O método `(/=)` está definido por defeito.

```
instance Eq Cor where
    Azul == Azul           = True
    Verde == Verde         = True
    Amarelo == Amarelo     = True
    Vermelho == Vermelho   = True
    _ == _                 = False
```

O tipo `Nat` também pode ser declarado como instância da classe Eq:

(==) de `Nat`

```
instance Eq Nat where
    (Suc n) == (Suc m)  = n == m
    Zero == Zero        = True
    _ == _              = False
```

O tipo `PontoC` com instância de Eq:

(==) de `Float`

```
instance Eq PontoC where
    (Pt x1 y1 c1) == (Pt x2 y2 c2) = (x1==x2) && (y1==y2)
                                     && (c1==c2)
```

(==) de `Cor`

Nota: **`(==)`** é uma função recursiva em `Nat`, mas não em `PontoC`.

Instâncias com restrições

Relembre a definição das árvores binárias.

```
data ArvBin a = Vazia
              | Nodo a (ArvBin a) (ArvBin a)
```

Como poderemos fazer o teste de igualdade para árvores binárias ?

Duas árvores são iguais se tiverem a mesma estrutura (a mesma forma) e se os valores que estão nos nodos também forem iguais.

Portanto, para fazer o teste de igualdade em `(ArvBin a)`, necessariamente, tem que se saber como testar a igualdade entre os valores que estão nos nodos, i.e., em a.

Só poderemos declarar `(ArvBin a)` como instância da classe `Eq` se a for também uma instância da classe `Eq`.

Este tipo de *restrição* pode ser colocado na declaração de instância, fazendo:

```
instance (Eq a) => Eq (ArvBin a) where
    Vazia == Vazia  = True
    (Nodo x1 e1 d1) == (Nodo x2 e2 d2) = (x1==x2) && (e1==e2)
                                                  && (d1==d2)
    _ == _  = False
```

(==) de a

(==) de (ArvBin a)

Instâncias derivadas de Eq

O testes de igualdade definidos até aqui implementam a **igualdade estrutural** *(dois valores são iguais quando resultam do mesmo construtor aplicado a argumentos também iguais)*.

Quando assim é pode-se evitar a declaração de instância se na declaração do tipo for acrescentada a instrução **deriving Eq**.

Exemplos: Com esta declarações, o Haskell deriva automáticamente declarações de instância de `Eq` (iguais às que foram feitas) para estes tipos.

```
data Cor = Azul | Amarelo | Verde | Vermelho
    deriving Eq
```

```
data Nat = Zero | Suc Nat
    deriving Eq
```

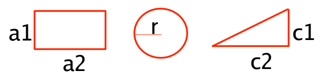
```
data PontoC = Pt {xx :: Float, yy :: Float, cor :: Cor}
    deriving Eq
```

```
data ArvBin a = Vazia
              | Nodo a (ArvBin a) (ArvBin a)
    deriving Eq
```


Mas, nem sempre a igualdade estrutural é a desejada.

Exemplo: Relembre o tipo de dados `Figura`:

```
data Figura = Rectangulo Float Float
            | Circulo Float
            | Triangulo Float Float
```



Neste caso queremos que duas figuras sejam consideradas iguais ainda que a ordem pela qual os valores são passados possa ser diferente.

```
instance Eq Figura where

    (Rectangulo x1 y1) == (Rectangulo x2 y2) =
                ((x1==x2) && (y1==y2)) || ((x1==y2)) && (x2==y1))

    (Circulo r1) == (Circulo r2) = r1==r2

    (Triangulo x1 y1) == (Triangulo x2 y2) =
                ((x1==x2) && (y1==y2)) || ((x1==y2)) && (x2==y1))
```


Exercícios:

- Considere a seguinte definição de tipo, para representar horas nos dois formatos usuais.

```
data Time = Am Int Int
          | Pm Int Int
          | Total Int Int
```

  Declare `Time` como instância da classe Eq de forma a que (==) teste se dois valores representam a mesma hora do dia, independentemente do seu formato.

- Qual o tipo principal da seguinte função:

```
lookup x ((y,z):yzs) | x /= y    = lookup x yzs
                     | othewise = Just z
lookup _ [] = Nothing
```

- Considere a seguinte declaração: `type Assoc a b = [(a,b)]`

  Será que podemos declarar `(Assoc a b)` como instância da classe Eq ?

Herança

O sistema de classes do Haskell também suporta a noção de **herança**.

Exemplo: Podemos definir a classe `Ord` como uma ***extensão*** da classe `Eq`.

```
-- isto é uma simplificação da classe Ord já pré-definida

class  (Eq a) => Ord a  where
    (<), (<=), (>=), (>) :: a -> a -> Bool
    max, min             :: a -> a -> a
```

A classe `Ord` **herda** todos os métodos de `Eq` e, além disso, establece um conjunto de operações de comparação e as funções máximo e mínimo.

Diz-se que `Eq` é uma ***superclasse*** de `Ord`, ou que `Ord` é uma ***subclasse*** de `Eq`.

Todo o tipo que é instância de `Ord` tem necessáriamente que ser instância de `Eq`.

Exemplo:

```
estaABProc :: Ord a => a -> ArvBin a -> Bool
estaABProc _ Vazia = False
estaABProc x (Nodo y e d) | x < y  = estaABProc x e
                          | x > y  = estaABProc x d
                          | x == y = True
```

A restrição `(Eq a)` não é necessária. Porquê ?

Herança múltipla

O sistema de classes do Haskell também suporta **herança múltipla**. Isto é, uma classe pode ter mais do que uma superclasse.

Exemplo: A classe `Real`, já pré-definida, tem a seguinte declaração

```
class  (Num a, Ord a) => Real a  where
    toRational :: a -> Rational
```

A classe `Real` herda todos os métodos da classe `Num` e da classe `Ord` e establece mais uma função.

NOTA: Na declaração dos tipos dos métodos de uma classe, é possível colocar restrições às variáveis de tipo, excepto à variável de tipo da classe que está a ser definida.

Exemplo:

```
class C a where
    m1 :: Eq b => (b,b) -> a -> a
    m2 :: Ord b => a -> b -> b -> a
```

O método `m1` impõe que b pertença à classe `Eq`, e o método `m2` impõe que b pertença a `Ord`.
Restrições à variável a, se forem necessárias, terão que ser feitas no contexto da classe, e nunca ao nível dos métodos.

A classe Ord

```
data Ordering = LT | EQ | GT
       deriving (Eq, Ord, Ix, Enum, Read, Show, Bounded)
```

```
class (Eq a) => Ord a where
    compare              :: a -> a -> Ordering
    (<), (<=), (>=), (>) :: a -> a -> Bool
    max, min             :: a -> a -> a

    -- Minimal complete definition: (<=) or compare
    -- using compare can be more efficient for complex types
    compare x y | x==y      = EQ
                | x<=y      = LT
                | otherwise = GT

    x <= y                  = compare x y /= GT
    x <  y                  = compare x y == LT
    x >= y                  = compare x y /= LT
    x >  y                  = compare x y == GT

    max x y   | x <= y      = y
              | otherwise   = x
    min x y   | x <= y      = x
              | otherwise   = y
```


Exemplos de instâncias de Ord

Exemplo:

```
instance Ord Nat where
    compare (Suc _) Zero = GT
    compare Zero (Suc _) = LT
    compare Zero Zero    = EQ
    compare (Suc n) (Suc m) = compare n m
```

Instâncias da classe `Ord` podem ser derivadas automaticamente. Neste caso, a relação de ordem é establecida com base na ordem em que os construtores são apresentados e na relação de ordem entre os parâmetros dos construtores.

Exemplo:

```
data AB a = V | NO a (AB a) (AB a)
      deriving (Eq,Ord)
```

```
ar1 = NO 1 V V
ar2 = NO 2 V V
```

```
> V < ar1
True
> ar1 < ar2
True
> (NO 4 ar1 ar2) < (NO 5 ar2 ar1)
True
> (NO 4 ar1 ar2) < (NO 3 ar2 ar1)
False
> (NO 4 ar1 ar2) < (NO 4 ar2 ar1)
True
```

Será que poderiamos não derivar Eq ?

As restrições às variáveis de tipo que são impostas pelo contexto, *propagam-se* ao logo do processo de inferência de tipos do Haskell.

Exemplo: Relembre a definição da função `quicksort`.

```
parte :: (Ord a) => a -> [a] -> ([a],[a])
parte _ [] = ([],[])
parte x (y:ys) | y < x = (y:as,bs)
               | otherwise = (as,y:bs)
   where (as,bs) = parte x ys
```

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) = let (l1,l2) = parte x xs
                   in (quicksort l1)++[x]++(quicksort l2)
```

Note como o contexto `(Ord a)` do tipo da função `parte` se propaga para a função `quicksort`.

A classe Show

A classe Show estabelece métodos para converter um valor de um tipo qualquer (que lhe pertença) numa string.

O interpretador Haskell usa o método **show** para apresentar o resultado dos seu cálculos.

```
class Show a where
    show      :: a -> String
    showsPrec :: Int -> a -> ShowS
    showList  :: [a] -> ShowS

    -- Minimal complete definition: show or showsPrec
    show x          = showsPrec 0 x ""
    showsPrec _ x s = show x ++ s
    showList []     = showString "[]"
    showList (x:xs) = showChar '[' . shows x . showl xs
       where showl []     = showChar ']'
             showl (x:xs) = showChar ',' . shows x . showl xs
```

```
type ShowS = String -> String
```

```
shows :: Show a => a -> ShowS
shows = showsPrec 0
```

A função showsPrec usa uma string como acumulador. É muito eficiente.

Exemplos de instâncias de Show

Exemplo:

```
natToInt :: Nat -> Int
natToInt Zero = 0
natToInt (Suc n) = 1 + (natToInt n)
```

```
instance Show Nat where
    show n = show (natToInt n)
```

```
> Suc (Suc Zero)
2
```

Instâncias da classe Show podem ser derivadas automaticamente. Neste caso, o método show produz uma string com o mesmo aspecto do valor que lhe é passado como argumento.

Exemplo: Se, em alternativa, tivessemos feito

```
data Nat = Zero | Suc Nat
    deriving Show
```

teriamos

```
> Suc (Suc Zero)
Suc (Suc Zero)
```

Exemplo:

```
instance Show Hora where
   show (AM h m) =  (show h) ++ ":" ++ (show m) ++ " am"
   show (PM h m) =  (show h) ++ ":" ++ (show m) ++ " pm"
```

```
> (AM 9 30)
9:30 am
```

```
> (PM 1 35)
1:35 pm
```


A classe Num

A classe Num está no topo de uma *hierarquia de classes (numéricas)* desenhada para controlar as operações que devem estar definidas sobre os diferentes tipos de números.

Os tipos `Int`, `Integer`, `Float` e `Double`, são instâncias desta classe.

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

    -- Minimal complete definition: All, except negate or (-)
    x - y          = x + negate y
    negate x       = 0 - x
```

A função `fromInteger` converte um `Integer` num valor do tipo `Num a => a`.

```
Prelude> :t 35
35 :: Num a => a
```

35 é na realidade `(fromInteger 35)`

```
Prelude>  35 + 2.1
37.1
```

Exemplos de instâncias de Num

Exemplo:

```
instance Num Nat where
    (+) = somaNat
    (*) = prodNat
    (-) = subtNat
    fromInteger = deInteger
    abs = id
    signum = sinal
    negate n = error "indefinido ..."
```

Note que Nat já pertence às classes Eq e Show.

```
prodNat :: Nat -> Nat -> Nat
prodNat Zero _    = Zero
prodNat (Suc n) m = somaNat m (prodNat n m)
```

```
subtNat :: Nat -> Nat -> Nat
subtNat n Zero          = n
subtNat (Suc n) (Suc m) = subtNat n m
subtNat Zero _          = error "indefinido ..."
```

```
sinal :: Nat -> Nat
sinal Zero    = Zero
sinal (Suc _) = Suc Zero
```

```
deInteger :: Integer -> Nat
deInteger 0     = Zero
deInteger (n+1) = Suc (deInteger n)
deInteger _     = error "indefinido ..."
```

```
somaNat :: Nat -> Nat -> Nat
somaNat Zero n    = n
somaNat (Suc n) m = Suc (somaNat n m)
```



```
tres = Suc (Suc (Suc Zero))
quatro = Suc tres
```

método da classe Num
somaNat

```
> tres + quatro
7
```

usa o método
show

```
> tres * quatro
12
```

método da classe Num
prodNat

```
> tres + 10
13
```

Nota: Não é possível derivar automaticamente instâncias da classe Num.

A classe Enum

A classe Enum establece um conjunto de operações que permitem *sequências aritméticas*.

```
class Enum a where
    succ, pred          :: a -> a
    toEnum              :: Int -> a
    fromEnum            :: a -> Int
    enumFrom            :: a -> [a]              -- [n..]
    enumFromThen        :: a -> a -> [a]         -- [n,m..]
    enumFromTo          :: a -> a -> [a]         -- [n..m]
    enumFromThenTo      :: a -> a -> a -> [a]    -- [n,n'..m]

    -- Minimal complete definition: toEnum, fromEnum
    succ                 = toEnum . (1+)        . fromEnum
    pred                 = toEnum . subtract 1 . fromEnum
    enumFrom x           = map toEnum [ fromEnum x ..]
    enumFromThen x y     = map toEnum [ fromEnum x, fromEnum y ..]
    enumFromTo x y       = map toEnum [ fromEnum x .. fromEnum y ]
    enumFromThenTo x y z = map toEnum [ fromEnum x, fromEnum y .. fromEnum z ]
```

Entre as instâncias desta classe contam-se os tipos: Int, Integer, Float, Char, Bool, ...

Exemplos:

```
Prelude> [2,2.5 .. 4]
[2.0,2.5,3.0,3.5,4.0]
```

```
Prelude> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
```


Exemplos de instâncias de Enum

Exemplo:

```
instance Enum Nat where
    toEnum = intToNat
        where intToNat :: Int -> Nat
              intToNat 0     = Zero
              intToNat (n+1) = Suc (intToNat n)

    fromEnum = natToInt
```

```
> [Zero, tres .. (tres * tres)]
[0,3,6,9]
> [Zero .. tres]
[0,1,2,3]
> [(Suc Zero), tres ..]
[1,3,5,7,9,11,13,15,17,19,21,23,25, ...
```

É possível derivar automaticamente instâncias da classe Enum, apenas em *tipos enumerados*.

Exemplo:

```
data Cor = Azul | Amarelo | Verde | Vermelho
    deriving (Enum,Show)
```

```
> [Azul .. Vermelho]
[Azul,Amarelo,Verde,Vermelho]
```

A classe Read

A classe Read estabelece funções que são usadas na conversão de uma string num valor do tipo de dados (instância de Read).

```
class Read a where
    readsPrec :: Int -> ReadS a
    readList  :: ReadS [a]

    -- Minimal complete definition: readsPrec
    readList   = ...
```

```
read :: Read a => String -> a
read s = case [x | (x,t) <- reads s, ("","") <- lex t] of
              [x] -> x
              []  -> error "Prelude.read: no parse"
              _   -> error "Prelude.read: ambiguous parse"
```

```
type ReadS a = String -> [(a,String)]
```

```
reads :: Read a => ReadS a
reads = readsPrec 0
```

lex é um *analisador léxico* definido no Prelude.

Podemos definir instâncias da classe Read que permitam fazer o *parser* do texto de acordo com uma determinada sintaxe. *(Mas isso não é tópico de estudo nesta disciplina.)*

Instâncias da classe Read podem ser derivadas automaticamente. Neste caso, a função read recebendo uma string que obedeça às regras sintácticas de Haskell produz o valor do tipo correspondente.

Exemplos:

```
data Time = Am Int Int
          | Pm Int Int
          | Total Int Int
  deriving (Show,Read)
```

```
data Nat = Zero | Suc Nat
    deriving Read
```

```
> read "Am 8 30" :: Time
Am 8 30
> read "(Total 17 15)" :: Time
Total 17 15
```

```
> read "Suc (Suc Zero)" :: Nat
2
```

```
> read "[2,3,6,7]" :: [Int]
[2,3,6,7]
> read "[Zero, Suc Zero]" :: [Nat]
[0,1]
```

É necessario indicar o tipo do valor a produzir.

Quase todos os tipos pré-definidos pertencem à classe Read.

Porquê ?

Declaração de tipos polimórficos com restrições nos parâmetros

Na declaração de um tipo algébrico pode-se exigir que os parâmetros pertençam a determinadas classes.

Exemplo:

```
data (Ord a) => STree a = Null
                        | Branch a (STree a) (STree a)
```

```
delSTree x Null = Null
delSTree x (Branch y e Null) | x == y = e
delSTree x (Branch y Null d) | x == y = d
delSTree x (Branch y e d)
               | x < y  = Branch y (delSTree x e) d
               | x > y  = Branch y e (delSTree x d)
               | x == y = let z = minSTree d
                          in Branch z e (delSTree z d)
```

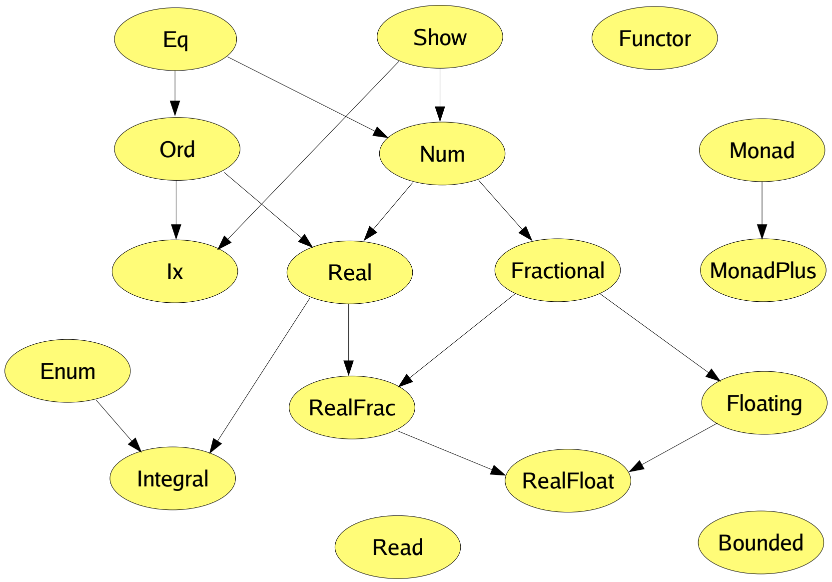
```
minSTree (Branch x Null _) = x
minSTree (Branch _ e _)    = minSTree e
```

Na declaração de tipos sinónimos também se podem impôr restricões de classes.

Exemplo:

```
type TAssoc a b = (Eq a) => [(a,b)]
```


Hierarquia de classes pré-definidas do Haskell




```
Prelude> :i Nome_da_Classe
```