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, estabelece 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 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
```

Será que poderiamos não derivar `Eq` ?

```
> 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
```

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` establece 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
```

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

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


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 ao 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 ..."
```

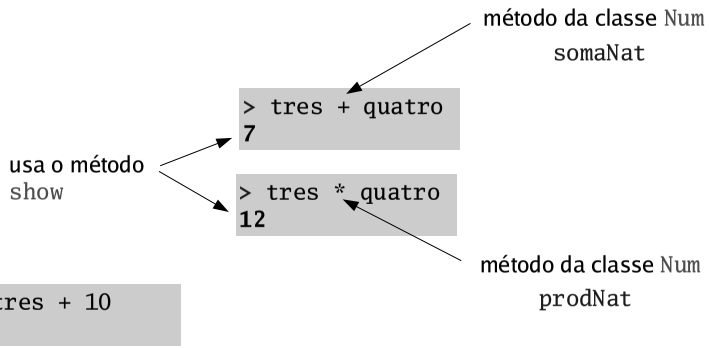
```
sinal :: Nat -> Nat
sinal Zero    = Zero
sinal (Suc _) = Suc Zero
```

```
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
```




```
> 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 = intNat
        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 ..]
[1,3,6,9,12,15,18,21,23,25, ...
```

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 establece 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)
```

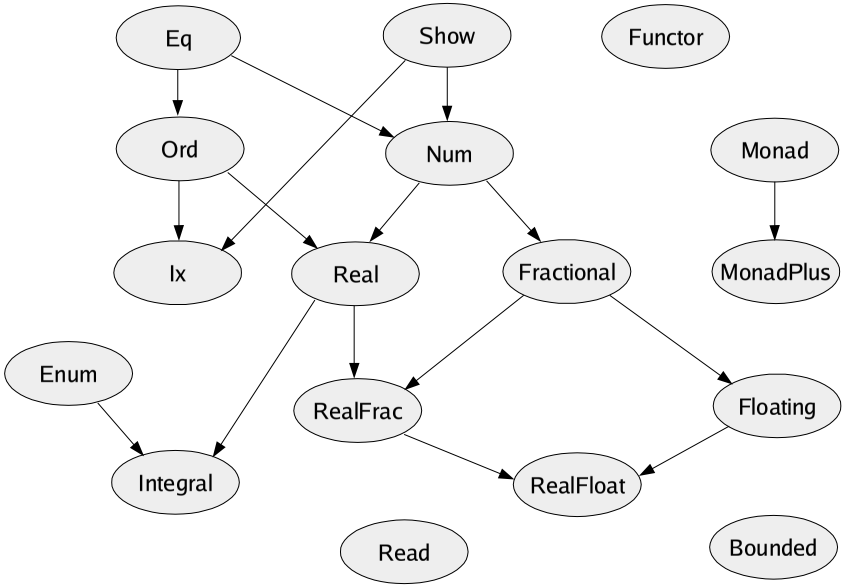
```
minSTree (Branch x Null _) = v
minSTree (Branch _ e _)    = minSTree e
```

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
```

Classes de Construtores de Tipos

Relembre os tipos paraméticos `(Maybe a)`, `[a]`, `(ArvBin a)`, `(Tree a)` ou `(ABin a b)`.

`Maybe`, `[ ]`, `ArvBin`, `Tree` e `ABin`, não são tipos, mas podem ser vistos como operadores sobre tipos – são **construtores de tipos**.

Exemplo: `Maybe` não é um tipo, mas `(Maybe Int)` é um tipo que resulta de aplicar o construtor de tipos `Maybe` ao tipo `Int`.

Em Haskell é possível definir classes de construtores de tipos. Um exemplo disso é a classe `Functor`:

```
class Functor f where
    fmap :: (a -> b) -> (f a -> f b)
```

Note que `f` não é um tipo.
`f a` e `f b` é que são tipos.

Exemplos:

```
instance Functor [] where
    fmap = map
```

```
instance Functor Maybe where
    fmap f Nothing  = Nothing
    fmap f (Just x) = Just (f x)
```

Note que o que se está a declarar como instância da classe `Functor` são construtores de tipos.

```
instance Functor ArvBin where
    fmap = mapAB
```


Definição de novas classes

Para além da hierarquia de classes pré-definidas, o Haskell permite definir novas classes.

Exemplo: Podemos definir a classe das *ordens parciais* da seguinte forma

```
class (Eq a) => OrdParcial a where
    comp :: a -> a -> Maybe Ordering     -- basta definir comp

    lt, gt, eq :: a -> a -> Maybe Bool
    lt x y = case (comp x y)
       of { Nothing -> Nothing ; (Just LT) -> Just True ; _ -> Just False }
    gt x y = case (comp x y)
       of { Nothing -> Nothing ; (Just GT) -> Just True ; _ -> Just False }
    eq x y = case (comp x y)
       of { Nothing -> Nothing ; (Just EQ) -> Just True ; _ -> Just False }

    maxi, mini :: a -> a -> Maybe a
    maxi x y = case (comp x y) of
                   Nothing  -> Nothing
                   Just GT  -> Just y
                   _        -> Just x
    mini x y = case (comp x y) of  Nothing  -> Nothing
                                   Just LT  -> Just x
                                   _        -> Just y
```

Nota: Repare nos diversos modos de escrever expressões case.

A relação de *inclusão de conjuntos* é um bom exemplo de uma relação de ordem parcial.

Exemplo: A noção de conjunto pode ser implementada pelo tipo

```
data (Eq a) => Conj a = C [a]   deriving Show
```

É necessário que se consiga fazer o teste de pertença.

```
instance (Eq a) => OrdParcial (Conj a) where
    comp (C u) (C v) = let p1 = u `contido` v
                           p2 = v `contido` u
                       in if p1 && p2 then Just EQ else
                          if p1       then Just LT else
                          if p2       then Just GT
                          else Nothing
       where
             contido :: (Eq a) => [a] -> [a] -> Bool
             contido xs ys = all (\x-> elem x ys) xs
```

```
> (C [2,1]) `gt` (C [7,1,5,2])
Just False
> (C [2,1,3]) `lt` (C [7,1,5])
Nothing
```

```
> (C [2,1,2,1]) `lt` (C [7,1,5,5,2])
Just True
> (C [3,3,5,1]) `eq` (C [5,1,5,3,1])
Just True
```


A noção de *função finita* establece um conjunto de associações entre *chaves* e *valores*, para um conjunto finito de chaves.

Exemplo: Podemos agrupar numa classe de construtores de tipos as opereções que devem estar definidas sobre funções finitas.

```
class FFinita ff where
    val :: (Eq a) => a -> (ff a b) -> Maybe b
    acr :: (Eq a) => (a,b) -> (ff a b) -> (ff a b)
    def :: (Eq a) => a -> (ff a b) -> Bool
    dom :: (Eq a) => (ff a b) -> [a]

    def x t = case (val x t) of
                   Nothing  -> False
                   (Just _) -> True
```

Exemplo: Tabelas implementando listas de associações (chave,valor) podem ser declaradas como instância da classe `FFinita`.

```
data (Eq a) => Tab a b = Tab [(a,b)]
  deriving Show
```

É possível usar o mesmo nome para o construtor de tipo e para o construtor de valores.

```
instance FFinita Tab where
    val x (Tab [])         = Nothing
    val x (Tab ((c,v):xs)) = if x==c then Just v
                                     else val x (Tab xs)

    acr (x,y) (Tab [])         = Tab [(x,y)]
    acr (x,y) (Tab ((c,v):t)) = if x==c
                                  then Tab ((x,y):t)
                                  else let (Tab w) = acr (x,y) (Tab t)
                                       in Tab ((c,v):w)

    dom (Tab t) = map fst t
```

Exercício:

- Defina um tipo de dados polimórfico que implemente listas de associações em árvores binárias e que possa ser instância da classe `FFinita`.
- Declare o construtor do tipo que acabou de definir como instância da classe `FFinita`.

Mónades

Na programação funcional, conceito de **mónade** é usado para sintetizar a ideia de computação.

Uma computação é vista como algo que se passa dentro de uma ***"caixa negra"*** e da qual conseguimos apenas ver os resultados.

Em Haskell, o conceito de mónade está definido como uma classe de construtores de tipos.

```
class Monad m where
    return :: a -> m a
    (>>=)  :: m a -> (a -> m b) -> m b      -- "bind"
    (>>)   :: m a -> m b -> m b             -- "sequence"
    fail   :: String -> m a

    -- Minimal complete definition: (>>=), return
    p >> q  = p >>= \ _ -> q
    fail s  = error s
```

- O termo `(return x)` corresponde a uma computação nula que retorna o valor `x`.
- O operador `(>>=)` corresponde de alguma forma à composição de computações.

A classe Monad

```
class Monad m where
    return :: a -> m a
    (>>=)  :: m a -> (a -> m b) -> m b      -- "bind"
    (>>)   :: m a -> m b -> m b             -- "sequence"
    fail   :: String -> m a

    -- Minimal complete definition: (>>=), return
    p >> q  = p >>= \ _ -> q
    fail s  = error s
```

- O termo `(return x)` corresponde a uma computação nula que retorna o valor `x`. `return` faz a transição do mundo dos valores para o mundo das computações.
- O operador `(>>=)` corresponde de alguma forma à composição de computações.
- O operador `(>>)` corresponde a uma composição de computações em que o valor devolvido pela primeira computação é ignorado.

`t :: m a` significa que `t` é uma computação que retorna um valor do tipo `a`.
Ou seja, `t` é um valor do tipo `a` com um efeito adicional captado por `m`.

Este efeito pode ser: uma acção de *input/output*, o tratamento de excepções, uma acção sobre o estado, etc.

Input / Output

Como conciliar o princípio de "computação por cálculo" com o input/output ?
Que tipos poderão ter as funções de input/output ?

Será que funções para ler um caracter do tecado, ou escrever um caracter no ecrã, podem ter os seguintes tipos ?

`lerChar :: Char` *É uma constante ?*

`escreveChar :: Char -> ()` *Como diferenciar da função* `f _ = ()` *?*

Em Haskell, existe pré-definido o **construtor de tipos** `IO`, e é uma instância da classe `Monad`.

Os tipos acima sugeridos estão errados. Essas funções estão pré-definidas e têm os seguintes tipos:

`getChar :: IO Char` — `getChar` é uma valor do tipo `Char` que pode resultar de alguma acção de input/output.

`putChar :: Char -> IO ()` — `putChar` é uma função que recebe um caracter e executa alguma acção de input/output, devolvendo `()`.