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 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 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 ..."
```

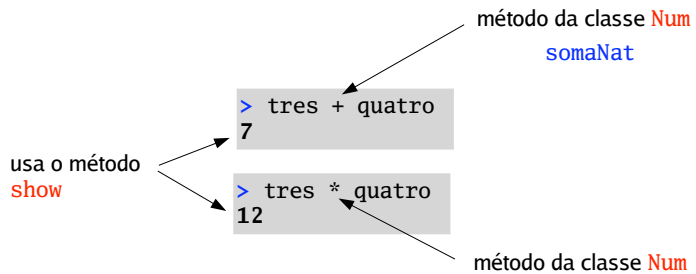
```
sinal :: Nat -> Nat
sinal Zero    = Zero
sinal (Suc _) = Suc Zero
```

```
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
```




```
> 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]
```