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

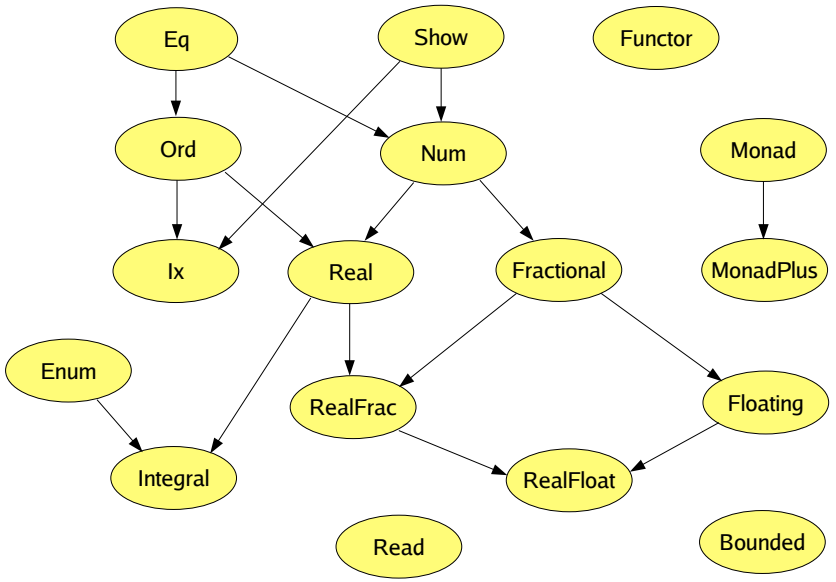
Na declaração de tipos sinónimos também se podem impôr restricões de classes.

**Exemplo:**

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



## Hierarquia de classes pré-definidas do Haskell




```
Prelude> :i Nome_da_Classe
```

## 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 x
                   _        -> Just y
    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 écran, 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 um 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 `()`.