

## 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) quando isso é possível.

```
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 do 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.

## A classe Read

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

Exemplos:

```
data Time = AM Int Int
          | PM Int Int
          | Total Int Int
          deriving (Read)
```

```
data Nat = Zero | Suc Nat
          deriving (Eq,Read)
```

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

```
> read "AM 8 30" :: Time
8:30 am
> read "(Total 17 15)" :: Time
17h15m
> read "Suc (Suc Zero)" :: Nat
2
> read "5+4" :: Int
*** Exception: Prelude.read: no parse
```

```
> read "[2,3,6,7]" :: [Int]
[2,3,6,7]
> read "[(AM 2 3), Total 5 6]" :: [Time]
[2:3 am,5h6m]
> read "[Zero, Suc Zero]" :: [Nat]
[0,1]
```

## Classes de construtores de tipos

Quando se faz uma declaração de um tipo algébrico, introduzem-se construtores de valores e construtores de tipos. Por exemplo,

```
data Maybe a = Nothing | Just a
```

**Maybe** é um construtor de tipo.

**Nothing** e **Just** são construtores de valores do tipo **Maybe a**

Em Haskell é possível definir classes de construtores de tipos.

Exemplo:

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

**f** é um construtor de tipo

Podemos declarar os construtores de tipos lista, BTree e Maybe como instância da classe Functor.

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

```
instance Functor BTree where
  fmap = mapBT
```

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

## Monads

O conceito de **mónade** é usado para sintetizar a ideia de **computação**. Uma computação é algo que se passa dentro de uma “caixa negra” e da qual conseguimos apenas ver os resultados.

**Monad** é uma classe de construtores de tipos do Haskell.

**return** corresponde a uma computação nula

**(>=)** compõe computações aproveitando o valor devolvido pela primeira para o cálculo da segunda.

**(>>)** compõe computações ignorando o valor devolvido pela primeira no cálculo da segunda.

```
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
```

**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, por exemplo, uma acção de IO.

# Input / Output

Como conciliar o princípio de “computação por cálculo” com o IO ?

Exemplo: Qual será o tipo de uma função `lerChar` que lê um caracter do teclado?

`lerChar :: Char`

?

Se assim fosse, `lerChar` seria uma constante do tipo `Char` !!!

- As funções do Haskell são funções matemáticas puras.
- Ler do teclado é um efeito lateral.
- Os programas interactivos têm efeitos laterais.
- As funções interactivas podem ser escritas em Haskell usando o construtor de tipos **IO**, para distinguir expressões puras de acções impuras que podem envolver efeitos laterais.
- (IO a)** é o tipo das acções de input/output que retornam um valor do tipo **a**.
- IO** é instância da classe **Monad**.
- A função que lê do teclado um caracter é `getChar :: IO Char`

`getChar` é um valor do tipo `Char` que resulta de uma acção de input/output.

# Algumas funções IO do Prelude

- Para **ler** do *standard input* (por omissão, o teclado):

```
getChar :: IO Char    lê um caracter;
getLine  :: IO String  lê uma string.
```

- Para **escrever** no *standard output* (por omissão, o ecrã):

```
putChar  :: Char -> IO ()  escreve um caracter;
putStr   :: String -> IO () escreve uma string;
putStrLn :: String -> IO () escreve uma string e muda de linha;
print    :: Show a => a -> IO () equivalente a (putStrLn . show)
```

- Para lidar com **ficheiros de texto**:

```
writeFile :: FilePath -> String -> IO ()  escreve uma string no ficheiro;
appendFile :: FilePath -> String -> IO () acrescenta no final do ficheiro;
readFile  :: FilePath -> IO String        lê o conteúdo do ficheiro para uma string.

type FilePath = String  é o nome do ficheiro (pode incluir a path no file system).
```

# Monad IO

O monade IO agrega os tipos de todas as computações onde existem acções de input/output.

- return :: a -> IO a** não faz nenhuma acção de IO. Apenas faz a conversão de tipo.
- (>>=) :: IO a -> (a -> IO b) -> IO b** compõe duas acções de IO podendo utilizar o valor devolvido pela primeira para o cálculo da segunda.
- (>>) :: IO a -> IO b -> IO b** compõe duas acções de IO de forma independente.

Exemplos: já definidos no Prelude

```
putStr :: String -> IO ()
putStr [] = return ()
putStr (x:xs) = (putChar x) >> (putStr xs)
```

```
getLine :: IO String
getLine = getChar >>= (\x-> if x=='\n'
                           then return []
                           else getLine >>= (\xs-> return (x:xs)) )
```

# Notação “do”

O Haskell fornece uma construção sintáctica (**do**) para escrever de forma simplificada cadeias de operações monádicas.

Exemplos: Podemos escrever

```
do e1
  e2      ou      do { e1; e2 }      em vez de      e1 >> e2

do x <- e1
  e2      em vez de      e1 >>= (\x -> e2)

do x1 <- e1
  x2 <- e2
  e3      em vez de      e1 >>= (\x1-> e2 >>= (\x2-> e3))
```

```
do e                                     = e
do e1; e2;...; en                       = e1 >> do e2;...; en
do x <- e1; e2;...; en                   = e1 >>= \ x -> do e2;...; en
do let declarações; e2;...; en          = let declarações in do e2;...; en
```

## Notação “do”

Exemplos:

```
putStr :: String -> IO ()
putStr [] = return ()
putStr (x:xs) = do putChar x
                  putStr xs
```

```
getline :: IO String
getline = do x <- getChar
            if x=='\n'
            then return []
            else do xs <- getline
                  return (x:xs)
```

Exemplo: Combinando “do” e “let”

```
test :: IO ()
test = do putStr "Escreva uma frase: "
         l <- getline
         let a = map toUpper l
             b = map toLower l
         putStrLn ("Maiúsculas: " ++ a)
         putStr ("Minúsculas: " ++ b)
```

```
> test
Escreva uma frase: aEIou
Maiúsculas: AEIOU
Minúsculas: aeiou
```

## Notação “do”

Exemplo: Defina a função dialogo que escreve no ecrã uma pergunta e recolhe a resposta dada.

```
dialogo :: String -> IO String
dialogo s = do putStr s
              r <- getline
              return r
```

```
dialogo' :: String -> IO String
dialogo' s = (putStr s) >> (getline >= (\r -> return r))
```

Exemplo: Defina a função questionario que recebe uma lista de questões e devolve a lista com as respostas dadas interactivamente .

```
questionario :: [String] -> IO [String]
questionario [] = return []
questionario (q:qs) = do r <- dialogo q
                        rs <- questionario qs
                        return (r:rs)
```

## Input / Output

Exemplo: Cálculo das raízes de um polinómio de 2º grau.

```
roots :: (Float,Float,Float) -> Maybe (Float,Float)
roots (a,b,c)
  | d >= 0 = Just ((-b + (sqrt d))/(2*a), (-b - (sqrt d))/(2*a))
  | d < 0 = Nothing
  where d = b^2 - 4*a*c
```

Camada interactiva:

```
calcRoots :: IO ()
calcRoots =
  do putStrLn "Calculo das raizes do polimomio a x^2 + b x + c"
     putStr "Indique o valor do ceoficiente a: "
     a <- getline
     putStr "Indique o valor do ceoficiente b: "
     b <- getline
     putStr "Indique o valor do ceoficiente c: "
     c <- getline
     case (roots (read a, read b, read c)) of
       Nothing      -> putStrLn "Não há raizes reais."
       (Just (r1,r2)) -> putStrLn ("As raizes são "++(show r1)++" e "++(show r2))
```

Float é instância da classe Read

## Input / Output

Uma maneira alternativa é usar a função **readIO** do Prelude

**readIO** :: Read a => String -> IO a      equivalente a (return . read)

```
calcROOTS :: IO ()
calcROOTS =
  do putStrLn "Calculo das raizes do polimomio a x^2 + b x + c"
     putStr "Indique o valor do ceoficiente a: "
     a <- getline
     a1 <- readIO a
     putStr "Indique o valor do ceoficiente b: "
     b <- getline
     b1 <- readIO b
     putStr "Indique o valor do ceoficiente c: "
     c <- getline
     c1 <- readIO c
     case (roots (a1,b1,c1)) of
       Nothing      -> putStrLn "Nao ha' raizes reais"
       (Just (r1,r2)) -> putStrLn ("As raizes sao "++(show r1)++" e "++(show r2))
```

## Input / Output

**Exemplo:** Carregar e descarregar uma base de dados de notas em ficheiro.

```
type Notas = [(Integer,String,Int)]
```

```
leFich :: IO ()
leFich = do file <- dialogo "Qual o nome do ficheiro ? "
          s <- readFile file
          let l = map words (lines s)
          print (geraNotas l)
```

```
geraNotas :: [[String]] -> Notas
geraNotas ([x,y,z]:t) = (read x, y, read z):(geraNotas t)
geraNotas _ = []
```

```
escFich :: Notas -> IO ()
escFich notas = do file <- dialogo "Qual o nome do ficheiro ? "
                  writeFile file (geraStr notas)
```

```
geraStr :: Notas -> String
geraStr [] = ""
geraStr ((x,y,z):t) = (show x) ++ ('\t':y) ++ ('\t':(show z)) ++ "\n" ++ (geraStr t)
```

Ficheiro de texto

12345	Ana	16
33333	Nuno	12
11111	Rui	18
22222	Ines	15

## Exercício

Implementar um jogo de adivinha com as seguintes regras:

- É gerado um número inteiro aleatório entre 0 e m.
- O jogador tenta adivinhar o número e o computador responde se o número é baixo, se o número é alto, ou se acertou, contabilizando o número de tentativas feitas pelo jogador até acertar.

Para gerar o número aleatório vai ser preciso importar a biblioteca **System.Random**, onde está a classe **Random** (dos tipos para os quais é possível gerar valores aleatórios), da qual **Int** é uma instância.

A função da classe que nos interessa neste caso é

```
randomRIO :: Random a => (a, a) -> IO a
```

que gera um valor aleatório do tipo a, dentro de um intervalo.

## Programas executáveis

- Para criar programas **executáveis** o compilador Haskell precisa de ter definido um módulo **Main** com uma função **main** que tem que ser de tipo **IO a**.
- A função **main** é o ponto de entrada no programa, pois é ela que é invocada quando o programa compilado é executado.
- A compilação de um programa Haskell, usando o GHC, pode ser feita executando no terminal do sistema operativo o seguinte comando:

```
ghc -o nome_do_executável -make nome_do_ficheiro_do_módulo_principal
```

**Exercício:** Crie um programa executável do jogo de adivinha que implementou.