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)
```

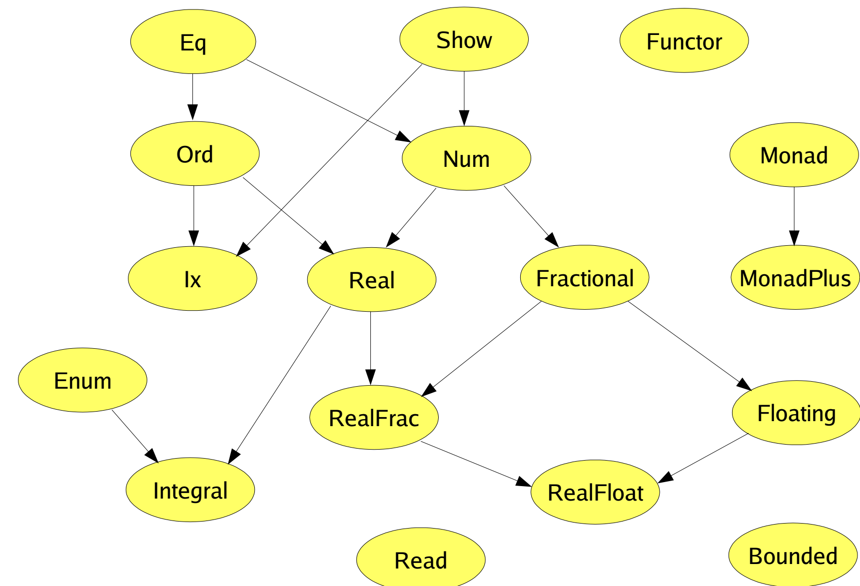
```
minSTree (Branch x Null _) = x
minSTree (Branch _ e _)    = minSTree e
```

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 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 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 `()`.

O mónade IO

O mónade IO agrupa os tipos de todas as computações onde existem acções de input/output.

`return :: a -> IO a` é a função que recebe um argumento x, não faz qualquer operação de IO, e retorna o mesmo valor x.

`(>>=) :: IO a -> (a -> IO b) -> IO b` é o operador que recebe como argumento um programa p, que faz alguma operações de IO e retorna um valor x, e uma função f que "transporta" esse valor para a próxima sequência de operações de IO.

`p >>= f` é o programa que faz as operações de IO correspondentes a p seguidas das operações de IO correspondentes a f x, retornando o resultado desta última computação.

Exemplo: As seguintes funções já estão pré-definidas.

```
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))
                      )
```


A notação "do"

O Haskell fornece uma construção sintática (do) para escrever de forma simplificada cadeias de operações mónadicas.

`e1 >> e2` pode ser escrito como `do { e1; e2 }` ou

```
do e1
   e2
```

`e1 >>= (\x -> e2)` pode ser escrito como

```
do x <- e1
   e2
```

`c1 >>= (\x1-> c2 >>= (\x2-> ... cn >>= (\xn-> return y) ...))`

pode ser escrito como

```
do x1 <- c1
   x2 <- c2
   ...
   xn <- cn
   return y
```

Mais formalmente:

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

A notação "do"

Exemplo: As funções pré-definidas putStr e getLine, usando a notação "do".

```
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: Misturando "do" e "let".

```
test :: IO ()
test = do x <- getLine
          let a = map toUpper x
              b = map toLower x
          putStr a
          putStr "\t"
          putStr b
          putStr "\n"
```

```
> test
aEIou
AEIOU  aeiou
>
```


Exemplos com IO

Exemplo:

```
expTrig :: IO ()
expTrig = do putStr "Indique um numero: "
             n <- getLine
             let x = ((read n)::Double)
                 s = sin x
                 c = cos x
             putStr ("O seno de "++n++" e' "++(show s)++['.','\n'])
             putStr ("O coseno de "++n++" e' "++(show c)++".\n")
```

```
> expTrig
Indique um numero: 2.5
O seno de 2.5 e' 0.5984721.
O coseno de 2.5 e' -0.8011436.
```

```
> expTrig
Indique um numero: 3.4.5
O seno de 3.4.5 e' *** Exception: Prelude.read: no parse
```

Exemplo:

Uma função que recebe uma listas de questões e vai recolhendo respostas para uma lista.

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

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

Ou, de forma equivalente:

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



```
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
```

```
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 <- return ((read a)::Float)
      putStr "Indique o valor do ceoficiente b: "
      b <- getLine
      b1 <- return ((read b)::Float)
      putStr "Indique o valor do ceoficiente c: "
      c <- getLine
      c1 <- return ((read c)::Float)
      case (roots (a1,b1,c1)) of
         Nothing       -> putStrLn "Nao ha' raizes reais."
         (Just (r1,r2)) -> putStrLn ("As raizes sao "++(show r1)
                                     ++" e "++(show r2))
```


Funções de IO do `Prelude`

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

```
getChar  :: IO Char
getLine  :: IO String
```

lê um caracter;
lê uma string (até se primir *enter*).

Para escrever no *standard ouput* (por defeito, o ecrã):

```
putChar  :: Char -> IO ()
putStr   :: String -> IO ()
putStrLn :: String -> IO ()
print    :: Show a => a -> IO ()
```

escreve um caracter;
escreve uma string;
escreve uma string e muda de linha;
equivalente a `(putStrLn . show)`

Para lidar com ficheiros de texto:

```
writeFile :: FilePath -> String -> IO ()
appendFile:: FilePath -> String -> IO ()
readFile  :: FilePath -> IO String
```

escreve uma string no ficheiro;
acrescenta no final do ficheiro;
lê o conteúdo do ficheiro para uma string.

```
type FilePath = String
```

é o nome do ficheiro (pode incluir a *path* no *file system*).

O módulo `IO` contém outras funções mais sofisticadas de manipulação de ficheiros.

O `Prelude` tem já definida a função `readIO`

```
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))
```

Exemplo:

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

texto = "1234\tPedro\t15\t17\n1111\tAna\t16\t13\n"
```

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

```
geraNotas :: [[String]] -> Notas
geraNotas ([x,y,z,w]:t) = let x1 = (read x)::Integer
                              z1 = (read z)::Int
                              w1 = (read w)::Int
                          in (x1,y,z1,w1):(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,w):t) = (show x) ++ ('\t':y) ++ ('\t':(show z)) ++
                        ('\t':(show w)) ++ "\n" ++ (geraStr t)
```


O mónade Maybe

A declaracão do construtor de tipos Maybe como instância da classe Monad é muito util para trabalhar com computações parciais, pois permite fazer a propagação de erros.

```
instance Monad Maybe where
    return x       = Just x
    (Just x) >>= f = f x
    Nothing >>= _  = Nothing
    fail _         = Nothing
```

Exemplo:

```
exemplo :: Int -> Int -> Int -> Maybe Int
exemplo a b c = do x <- return a
                   y <- return b
                   z <- divide x y
                   w <- soma c z
                   return w
```

Podemos simplificar ?

```
divide :: Int -> Int -> Maybe Int
divide _ 0 = Nothing
divide x y = Just (div x y)
```

```
soma :: Int -> Int -> Maybe Int
soma x y = Just (x+y)
```


Módulos

Um programa Haskell é uma colecção de **módulos**. A organização de um programa em módulos cumpre dois objectivos:

- criar componentes de software que podem ser usadas em diversos programas;
- dar ao programador algum control sobre os identificadores que podem ser usados.

Um módulo é uma declaração "gigante" que obdece à seguinte sintaxe:

module *Nome (entidades_a_exportar)* **where**

declarações de importações de módulos

declarações de: tipos, classes, instâncias, assinaturas, funções, ...
(por qualquer ordem)

Cada módulo está armazenado num ficheiro, geralmente com o mesmo nome do módulo, mas isso não é obrigatório.

Na declaração de um módulo:

- pode-se indicar explicitamente o conjunto de tipos / construtores / funções / classes que são exportados (i.e., visíveis do exterior)

  *Aos vários items que são exportados ou importados chamaremos **entidades**.*

- por defeito, se nada for indicado, todas as declarações feitas do módulo são exportadas;
- é possível exportar um tipo algébrico com os seus construtores fazendo, por exemplo: `ArvBin(Vazia, Nodo)`, ou equivalentemente, `ArvBin(..)`;
- também é possível exportar um tipo algébrico e não exportar os seus construores, ou exportar apenas alguns;
- os métodos de classe podem ser exportados seguindo o estilo usado na exportação de construtores, ou como funções comuns;
- declarações de instância são sempre exportadas e importadas, por defeito;
- é possível exportar entidades que não estão directamente declaradas no módulo, mas que resultam de alguma importação de outro módulo.

Qualquer entidade visível no módulo é passível de ser exportada por esse módulo.

Na importação de um módulo por outro módulo:

- é possível fazer a importação de todas as entidades exportadas pelo módulo fazendo

```
import Nome_do_módulo
```

- é possível indicar explicitamente as entidades que queremos importar, fazendo

```
import Nome_do_módulo (entidades a importar)
```

- é possível indicar selectivamente as entidades que não queremos importar (importa-se tudo o que é exportado pelo outro módulo excepto o indicado)

```
import Nome_do_módulo hiding (entidades a não importar)
```

- é possível fazer com que as entidades importadas sejam referenciadas indicando o módulo de onde provêm como prefixo (seguido de '.') fazendo

```
import qualified Nome_do_módulo (entidades a importar)
```

(Pode ser util para evitar colisões de nomes, pois é ilegal importar entidades diferentes que tenham o mesmo nome. Mas se for o mesmo objecto que é importado de diferentes módulos, não há colisão. Uma entidade pode ser importada via diferentes caminhos sem que haja conflitos de nomes.)

Um exemplo com módulos

Considere os módulos: `Listas`, `Arvores`, `Tempo`, `Horas` e `Main`, que pretendem ilustrar as diferentes formas de exportar e importar entidades.

```
module Listas where

soma [] = 0
soma (x:xs) = x + (soma xs)

conta = length

naLista x [] = False
naLista x (y:ys) = if x==y then True
                           else naLista x ys

mult = product

cauda (_:xs) = xs
```



```
module Arvores(ArvBin(Vazia,Nodo), naArv, soma, mult) where

data ArvBin a = Vazia
              | Nodo a (ArvBin a) (ArvBin a)
  deriving Show

conta Vazia = 0
conta (Nodo _ e d) = 1 + (conta e) + (conta d)

soma Vazia = 0
soma (Nodo x e d) = x + (soma e) + (soma d)

mult Vazia = 1
mult (Nodo x e d) = x * (mult e) * (mult d)

naArv :: (Eq a) => a -> ArvBin a -> Bool
naArv _ Vazia = False
naArv x (Nodo y e d) | x==y      = True
                     | otherwise = (naArv x e) || (naArv x d)
```



```
module Tempo(Time, horas, minutos, meioDia, cauda) where

import Listas

data Time = Am Int Int
          | Pm Int Int
          | Total Int Int   deriving Show

hValida (Total h m) = 0<=h && h<24 && 0<=m && m<60
hValida (Am h m)    = 0<=h && h<12 && 0<=m && m<60
hValida (Pm h m)    = 0<=h && h<12 && 0<=m && m<60

horas (Am h m)    = h
horas (Pm h m)    = h + 12
horas (Total h m) = h

minutos (Am h m)    = m
minutos (Pm h m)    = m
minutos (Total h m) = m

meioDia = (Total 12 00)

ex = cauda “experiencia”
```

```haskell
module Horas(Hora(..), Tempo(manha)) where

data Hora = AM Int Int
          | PM Int Int


class Tempo a where
    manha :: a -> Bool
    tarde :: a -> Bool
    tarde t = not (manha t)


instance Tempo Hora where
    manha (AM _ _) = True
    manha (PM _ _) = False
```



```haskell
module Main where

import Arvores (ArvBin(..), soma, naArv)

import qualified Listas (soma, mult, conta)

import Tempo

import Horas

import Char hiding (toUpper, isDigit)

arv1 = Nodo 5 (Nodo 3 Vazia (Nodo 4 Vazia Vazia))
              (Nodo 2 (Nodo 1 Vazia Vazia) Vazia)

lis1 = [1,2,3,4]

minTotal :: Time -> Int
minTotal t = (horas t)*60 + (minutos t)

testeC = cauda lis1

toUpper :: Num a => ArvBin a -> ArvBin a
toUpper Vazia = Vazia
toUpper (Nodo x e d) = Nodo (x*x) (toUpper e) (toUpper d)

test = map toLower "tesTAnDo"
```


Após carregar o módulo Main, analise o comportamento do interpretador.

```
*Main> soma arv1
15
*Main> mult arv1
   Variable not in scope: `mult'
*Main> conta arv1
   Variable not in scope: `conta'
*Main> Listas.soma lis1
10
*Main> mult lis1
   Variable not in scope: `mult'
*Main> Listas.mult lis1
24
```

```
*Main> testeC
[2,3,4]
*Main> hValida meioDia
   Variable not in scope: `hValida'
```

```
*Main> isDigit 'e'
   Variable not in scope: `isDigit'
*Main> isAlpha 'e'
True
*Main> toUpper arv1
Nodo 25 (Nodo 9 Vazia (Nodo 16 Vazia Vazia))
        (Nodo 4 (Nodo 1 Vazia Vazia) Vazia)
*Main> test
"testando"
```

```
*Main> minTotal meioDia
720
*Main> minTotal (Am 9 30)
   Data constructor not in scope: `Am'
*Main> manha (AM 9 30)
True
*Main> tarde (PM 17 15)
   Variable not in scope: `tarde'
```


Compilação de programas Haskell

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 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 *Glasgow Haskell Compiler*, pode ser feita executando na shell do sistema operativo o seguinte comando:

ghc -o *nome_do_executável* **--make** *nome_do_ficheiro_do_módulo_principal*

Exemplo: Usando o último exemplo para testar a compilação de programas definidos em vários módulos, podemos acrescentar ao módulo Main a declaração

```haskell
main = print "OK"
```

Assumindo que este módulo está guardado no ficheiro **Main.hs** podemos fazer a compilação assim:

```
ghc -o testar --make Main
```

Exemplo: Assumindo que o módulo do próximo slide está no ficheiro **roots.hs**, podemos gerar um executável (chamado raizes) fazendo

```
ghc -o raizes --make roots
```

```
module Main where

main :: IO ()
main = do calcRoots
          putStrLn "Deseja continuar (s/n) ? "
          x <- getLine
          case (head x) of
               's' -> main
               'S' -> main
               _   -> putStrLn "\n FIM."

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

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
```


Tipos Abstractos de Dados

A quase totalidade dos tipos de dados que vimos até aqui são **tipos concretos de dados**, dado que se referem a uma estrutura de dados concreta fornecida pela linguagem.

Exemplos:

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

```
type TB = [(Integer,String)]
```

`(ArvBin a)` e `TB` são dois tipos concretos. Sabemos como são constituidos os valores destes tipos e podemos extrair informação ou contruir novos valores, por manipulação directa dos construtores de valores destes tipos.

Em contraste, os **tipos abstractos de dados** não estão ligados a nenhuma representação particular. Em vez disso, eles são definidos implicitamente através de um conjunto de operações utilizadas para os manipular.

Exemplo: O tipo `(IO a)` é um tipo abstracto de dados. Não sabemos de que forma são os valores deste tipo. Apenas conhecemos um conjunto de funções para os maniputar.

Tipos Abstractos de Dados

As assinaturas das funções do tipo abstracto de dados e as suas especificações constituem o ***interface*** do tipo abstracto de dados. Nem a estrutura interna do tipo abstracto de dados, nem a implementação destas funções são visíveis para o utilizador.

Dada a especificação de um tipo abstracto de dados, as operações que o definem poderão ter *diferentes implementações*, dependendo da estrutura usada na representação interna de dados e dos algoritmos usados.

A utilização de tipos abstractos de dados trás benefícios em termos de **modularidade** dos programas. Alterações na implementação das operações do tipo abstracto não afecta outras partes do programa desde que as operações mantenham o seu tipo e a sua especificação.

Em Haskell, a construção de tipos abstractos de dados é feita utilizando módulos.

O módulo aonde se implementa o tipo abstracto de dados deve exportar apenas o nome do tipo e o nome das operações que constituem o seu interface. A representação do tipo fica assim escondida dentro do módulo, não sendo visível do seu exterior.

Deste modo, podemos mais tarde alterar a representação do tipo abstracto sem afectar os programas que utiliza esse tipo abstracto.

Stacks (pilhas)

Uma **Stack** é uma colecção homegénea de items que implementa a noção de pilha, de acordo com o seguinte interface:

`push :: a -> Stack a -> Stack a`	coloca um item no topo da pilha
`pop :: Stack a -> Stack a`	remove o item do topo da pilha
`top :: Stack a -> a`	dá o item que está no topo da pilha
`stackEmpty :: Stack a -> Bool`	testa se a pilha está vazia
`newStack :: Stack a`	cria uma pilha vazia

Os items da Stack são removidos de acordo com a estratégia **LIFO** (Last In First Out).

Exemplo: