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 `()`.

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))
```


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 écran):

```
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.


```
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))
```


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 obedece 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 construtores, 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.