

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 algumas 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))
                           )
```

153

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
>
```

155

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:

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

154

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 ("0 seno de "++n++" e' "++(show s)+[".', '\n'"])
            putStr ("0 coseno de "++n++" e' "++(show c)+[".', '\n']")
```

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

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

156

Exemplo:

Uma função que recebe uma lista 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))
```

157

```
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 coeficiente a: "
     a <- getLine
     a1 <- return ((read a)::Float)
     putStr "Indique o valor do coeficiente b: "
     b <- getLine
     b1 <- return ((read b)::Float)
     putStr "Indique o valor do coeficiente 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))
```

159

Funções de IO do Prelude

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

```
getChar :: IO Char    lê um caracter;
getLine :: IO String  lê uma string (até se primir enter).
```

Para escrever no *standard output* (por defeito, 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).
```

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

158

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 coeficiente a: "
     a <- getLine
     a1 <- readIO a
     putStr "Indique o valor do coeficiente b: "
     b <- getLine
     b1 <- readIO b
     putStr "Indique o valor do coeficiente 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))
```

160

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)
```

161

Módulos

Um programa Haskell é uma coleção 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.

163

O mónade Maybe

A declaração do construtor de tipos **Maybe** como instância da classe **Monad** é muito útil 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)
```

162

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 itens 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.

164