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 dasta 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 ecrá):

```
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))
```