Input/Output em Haskell

Autor: Francisco Soares (xfrancisco.soares@gmail.com)

Modificações: Luís Gabriel Lima (lgnfl)

Márcio Lopes Cornélio

(elaborado a partir dos *slides* de **André Santos** e **Sérgio Soares**)

Até aqui

- Programas (funções) auto-contidos
  - sem muita interação com o usuário

- Mas programas também devem
  - ler e escrever no terminal
  - ler e escrever arquivos
  - controlar dispositivos

ou seja, realizar operações de entrada e saída

O tipo `IO t`

- Todo tipo **IO a** é uma ação de I/O.

- Exemplo – imprimir uma String:

```
writefoo :: IO ()
writefoo = putStrLn "foo"
```

- Somente digitar isso não produz nada, mas ao chamar **`writefoo`**, "foo" é impresso.

Haskell e efeitos colaterais

- Mas o fato de Haskell ser *puramente funcional* não impediria efeitos colaterais?
  - Por exemplo, entrada e saída.
- Quase!

O tipo `IO t`

- Imagine tais tipos como programas que executam entrada/saída (IO) e retornam um valor do tipo `t`

- Lê uma linha do teclado

```
getLine :: IO String
```

O tipo `IO t`

- Escreve uma String na tela.

```
putStr :: String -> IO ()
```

o resultado desta interação tem tipo `()`, que é uma tupla vazia. Neste caso significa dizer que a função não retorna nenhum resultado interessante, apenas faz I/O.

O tipo `IO t`

- Lê um caracter do teclado.

```
getChar :: IO Char
```

- Caso especial da função **`putStr`**, que insere uma quebra de linha no final.

```
putStrLn :: String -> IO ()
```

Tipo ()

Tipo com um único elemento: ()

Um valor deste tipo não possui informação útil

Útil no caso de realização de IO porque há programas de IO que têm como importante apenas a ação de IO, não o resultado que a ação retorna

Sequenciando ações de IO

- A operação **do**

```
putStrLn :: String -> IO ()
putStrLn str = do putStr str
                  putStr "\n"
```

Sequenciando ações de IO

```
put4times :: String -> IO ()
put4times str = do putStr str
                   putStr str
                   putStr str
                   putStr str
```

Sequenciando ações de IO

- A operação **`if-then`**

```
putNtimes :: Int -> String -> IO ()
putNtimes n str
  = if n <= 1
       then putStr str
       else do putStr str
               putNtimes (n-1) str
```

Sequenciando ações de IO

- Lendo informações do teclado

```
getNput :: IO ()
getNput = do line <- getLine
             putStr line
```

A linha '**`line <-`**' **nomeia** a saída da função **`getLine`**

  - Mas '*id* < -' associa o nome *id* ao retorno da função getLine

Exemplos: sequência de ações de IO

```
reverse2lines:: IO ()
reverse2lines
  = do line1 <- getLine
       line2 <- getLine
       putStrLn (reverse line2)
       putStrLn (reverse line1)
```

Exemplos: sequência de ações de IO

```
main = do
    return ()
    return "HAHAHA"
    line <- getLine
    return "BLAH BLAH BLAH"
    return 4
    putStrLn line
```

Exemplos: sequência de ações de IO

```
main = do
    line <- getLine
    if null line
        then return ()
        else (do
            putStrLn $ reverseWords line
            main)
reverseWords :: String -> String
reverseWords = unwords . map reverse. words
```

Monads e IO

- O tipo **`IO t`** é da classe **`Monad`**, uma classe que define funções para sequenciamento de ações.
- Para o propósito desta aula, devemos só usar desta classe o tipo **`IO`**, mas há outros, por exemplo:
  - data Maybe a = Nothing | Just a
  - []

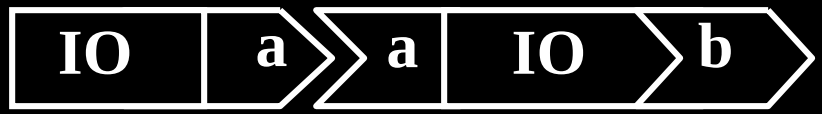
Outra abordagem

A operação ‘bind’, simbolizada por **`>>=`** dá sequência a duas operações, uma após a outra.

```
(>>=)::(Monad m) => m a->(a -> m b)->m b
```

Esta operação combina um **`IO a`** com uma função que pega o resultado dessa expressão (do tipo **`a`**) e retorna algo de tipo **`IO b`**.

Podemos combinar as expressões, passando o resultado da primeira como primeiro argumento da segunda.



Outras Interações

- O operador **>>** é igual ao **>>=**, mas ignora o resultado da primeira para a segunda interação

```
(>>) :: (Monad m) => m a -> m b
                  -> m b
```

- Retorna um valor do tipo **m a** (converte do tipo **a** para o tipo **IO a**, por exemplo)

```
return :: (Monad m)=> a -> m a
```

O tipo `IO t`

- Lembrem-se: em todo caso que um operador funcionar para um ***Monad***, funcionará para `IO`.

Exemplo

```
main :: IO()
main = putStr "Digite seu nome:"  >>
       getLine >>=
       \st ->
         putStr “Ao contrario e':"  >>
         putStr (reverse st)
```

Exemplo

```
main :: IO()
main = do putStr "Digite seu nome:"
          st <- getLine
          putStr “Ao contrario e':"
          putStr (reverse st)
```

Resumo de funções para `IO t`

```
getLine :: IO String

getChar :: IO Char

putStr :: String -> IO ()

putStrLn :: String -> IO ()

(>>=) :: IO a -> (a -> IO b) -> IO b

(>>) :: IO a -> IO b -> IO b

return :: a -> IO a
```

Manipulação de arquivos

- Haskell manipula arquivos através do tipo **`FilePath`**, um tipo sinônimo.

```
type FilePath = String
```

Manipulação de arquivos

- Leitura de arquivos:

```
readFile:: FilePath -> IO String
```

- Escrever em arquivos:

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

- Anexar a arquivos:

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

Exemplo

```
main :: IO ()
main =
  putStrLn "Escrevendo" >>
  writeFile "a.txt" "Hello\nworld" >>
  appendFile "a.txt" "\nof\nHaskell"
  >> putStrLn "Lendo o arquivo" >>
  readFile "a.txt" >>=
  \x -> putStrLn x
```

Mais informações

- Monads:

- http://learnyouahaskell.com/functors-applicative-functors-and-monoids

- http://learnyouahaskell.com/a-fistful-of-monads

- http://www.haskell.org/haskellwiki/Monad

- System.IO:

- http://lambda.haskell.org/hp-tmp/docs/2011.2.0.0/ghc-doc/libraries/haskell2010-1.0.0.0/System-IO.html

Exercício

- Escreva uma função shorten :: String -> String que encurta links do YouTube. Em seguida, faça um programa em Haskell que lê de um arquivo url.in várias URLs do YouTube e para cada uma imprime na saída padrão a respectiva URLs curta.

Solução

```
shorten :: String -> String
shorten url = "http://youtu.be/" ++ getVideoId url

getVideoId :: String -> String
getVideoId url = removeFrom x '&' []
  where
    x = removeWord "http://www.youtube.com/watch?v=" url

removeWord :: String -> String -> String
removeWord _ [] = []
removeWord [] x = x
removeWord (x:xs) (y:ys)
    | x == y     = removeWord xs ys
    | otherwise = ys

removeFrom :: String -> Char -> String -> String
removeFrom [] _ acc = acc
removeFrom (x:xs) c acc
    | x == c     = acc
    | otherwise = removeFrom xs c (acc ++ [x])
```

Solução

```
Import System.IO

loop :: Handle -> IO ()
loop handle = do
    test <- hIsEOF handle
    if test
        then return ()
        else do
          url <- hGetLine handle
          let shortUrl = shorten url
          putStrLn shortUrl
          loop handle

main :: IO ()
main = do
    handle <- openFile "url.in" ReadMode
    loop handle
```