Entrada e Saída

João Marcelo Uchôa de Alencar

Universidade Federal do Ceará - Quixadá

23 de Junho de 2017

Introdução

- As funções funções em Haskell que trabalhamos até agora não alteram estado.
- Em outras palavras, sempre que uma função é executada com os mesmos parâmetros, podemos ter certeza que o retorno será o mesmo.
- Não há atualização de variáveis, endereços de memórias, etc.
- Sempre uma nova estrutura é retornada.
- Não há efeitos colaterais.
- Podemos ter funções *impuras* para alterar o estado da entrada e saída

Introdução

- Até agora fizemos tudo através do *ghci*.
- Porém está na hora de evoluirmos para fazer programas de verdade.
- Vamos o usar o *ghc*.

```
-- olamundo.hs
main = putStrLn "Olá Mundo"
```

```
$ ghc --make olamundo
[1 of 1] Compiling Main ( olamundo.hs, olamundo.o )
Linking olamundo ...
$ ./olamundo
Olá Mundo
```

Entrando nos Detalhes...

```
ghci> :t putStrLn
putStrLn :: String -> IO ()
ghci> :t putStrLn "Olá Mundo"
putStrLn "Olá Mundo" :: IO ()
```

- *putStrLn* recebe uma cadeia de caracteres e retorna uma ação de E/S.
- Uma ação de E/S é uma execução que altera o estado dos dispositivos de E/S.
- No caso como é uma escrita na tela, o retorno não tem tipo, portanto é uma tupla vazia ().
- As ações de E/S são executadas na definição da função *main*.

Múltiplas E/S

```
main = do
    putStrLn "Olá, como você se chama?"
    nome <- getLine
    putStrLn ("Olá " ++ nome ++ ", você se garante!!!")
```

O *do* permite informar várias ações de E/S para *main*.

```
ghci> :t getLine
getLine :: IO String
```

getLine é uma ação de E/S que retorna uma *String*.
Ao ser executada duas vezes, *getLine* não garante entregar o mesmo resultado.

```
-- Funciona?
nome = "Olá, meu nome é " ++ getLine
```

Múltiplas E/S

```
main = do
    foo <- putStrLn "Olá, qual é seu nome?"
    nome <- getLine
    putStrLn ("Olá " ++ nome ++ ", você se garante!")
```

- *foo* terá tipo (), o que não é muito útil.
- Em um bloco *do*, toda ação pode ser ligada a um nome, exceto a última.

```
-- Isto faz sentido?
nome = getLine
```

Onde as ações podem executar?

- Dentro da atribuição de *main*.
- Dentro uma ação maior criada a partir de um bloco *do*.
- Se forem invocadas diretamente no *ghci*.

Invocando Pureza

```
import Data.Char

main = do
    putStrLn "Qual seu primeiro nome?"
    primeiroNome <- getLine
    putStrLn "Qual seu segundo nome?"
    ultimoNome <- getLine
    let maiorPrimeiroNome = map toUpper primeiroNome
        maiorUltimoNome = map toUpper ultimoNome
    putStrLn $ "Olá " ++ maiorPrimeiroNome ++ " "
                ++ maiorUltimoNome ++ ", como está você?"
```

Inversão de Linhas

```
main = do
    linha <- getLine
    if null linha
        then return ()
        else do
            putStrLn $ inverterPalavras linha
            main

inverterPalavras :: String -> String
inverterPalavras = unwords . map reverse . words
```

O *return* não retorna!

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

- O *return* permite personalizar a ação de retorno.
- Entretanto, não interrompe o fluxo das ações.

putStr

```
main = do putStr "Programação "
          putStr "Funcional "
          putStrLn "Quixadá "
$ runhaskell putstr_test.hs
Programação Funcional Quixadá
```

putChar

```
main = do putChar 't'
          putChar 'e'
          putChar 'h'
$ runhaskell putchar_test.hs
teh
-- Definição recursiva de putStr
putStr :: String -> IO ()
putStr [] = return ()
putStr (x:xs) = do
    putChar x
    putStr xs
```

print

```
-- Invoca o comportamento show do tipo.
main = do print True
          print 2
          print "haha"
          print 3.2
          print [3,4,3]
$ runhaskell print_test.hs
True
2
"haha"
3.2
[3,4,3]
```

getChar

```
main = do
    c <- getChar
    if c /= ' '
       then do
          putChar c
          main
       else return ()
```

when

```
import Control.Monad

main = do
    c <- getChar
    when (c /= ' ') $ do
        putChar c
        main
```

sequence

```haskell
-- No lugar disto:
main = do
    a <- getLine
    b <- getLine
    c <- getLine
    print [a,b,c]
-- Podemos escrever isto:
main = do
    rs <- sequence [getLine, getLine, getLine]
    print rs
```

mapM e mapM_

```
ghci> mapM print [1,2,3]
1
2
3
[(),(),()]
ghci> mapM_ print [1,2,3]
1
2
3
```

forever

```haskell
import Control.Monad
import Data.Char

main = forever $ do
    putStr "Give me some input: "
    l <- getLine
    putStrLn $ map toUpper l
```

forM

```
import Control.Monad

main = do
    cores <- forM [1,2,3,4] (\a -> do
          putStrLn $ "Qual cor você associa com o número: "
                     ++ show a ++ "?"
          cor <- getLine
          return cor)
    putStrLn "As cores que você associou com 1, 2, 3 e 4 são: "
    mapM putStrLn cores
```

Problema dos Elefantes

Escreva uma função `elefantes :: Int -> IO ()` tal que, por exemplo, elefantes 5 imprime os seguintes versos:

```
Se 2 elefantes incomodam muita gente,
3 elefantes incomodam muito mais!
Se 3 elefantes incomodam muita gente,
4 elefantes incomodam muito mais!
Se 4 elefantes incomodam muita gente,
5 elefantes incomodam muito mais!
```

Sugestão: utilize a função `show :: Show a => a -> String` para converter um inteiro numa cadeia de caracteres; pode ainda reutilizar a função `sequence` para executar uma lista de ações.

Entendendo o que é um *pipe* no mundo Unix

```haskell
-- Ler toda uma linha por vez
import Control.Monad
import Data.Char

main = forever $ do
    l <- getLine
    putStrLn $ map toUpper l
-- Versão Preguiçosa
import Data.Char

main = do
    contents <- getContents
    putStr (map toUpper contents)
```

Selecionando Linhas Pelo Tamanho

```
main = do
    contents <- getContents
    putStr (menorQueDez contents)

menorQueDez :: String -> String
menorQueDez input =
      let linhas = lines input
          linhascurtas = filter
                  (\linha -> length linha < 10) linhas
          resultado = unlines linhascurtas
      in resultado
```

Usando *interact* para filtragem.

```
main = interact menorQueDez

menorQueDez :: String -> String
menorQueDez input =
      let linhas = lines input
          linhascurtas = filter
                  (\linha -> length linha < 10) linhas
          resultado = unlines linhascurtas
      in resultado

-- Simplificando ao máximo
main = interact $
        unlines . filter ((<10) . length) . lines
```

Filtrando Palíndromes

```
main = interact palindromes

palindromes = unlines .
              map (\xs -> if ePalindrome xs
                          then "palíndrome"
                          else "não é palíndrome") .
                          lines
              where ePalindrome xs = xs == reverse xs
```

E os Arquivos?

- Até agora, mesmo usando *pipes*, a entrada e saída foi através do terminal.
- Cada processo no Linux, seja qual for a linguagem, tem três arquivos virtuais: *stdin*, *stdout* e *stderr*.
- Usaremos ações de E/S parecidas com a que usamos até agora nos fluxos padrão.

Abrindo um Arquivo

```haskell
import System.IO

-- openFile :: FilePath -> IOMode -> IO Handle
-- type FilePath = String
-- data IOMode = ReadMode | WriteMode |
--               AppendMode | ReadWriteMode
main = do
    handle <- openFile "arquivo_texto.txt" ReadMode
    contents <- hGetContents handle
    putStr contents
    hClose handle
```

O que é o *handle* ?

- O *handle* é um marcador que indica até onde foi lido (ou escrito) em um arquivo.
- Assim como no terminal, o arquivo não é lido de uma vez para a memória por *hGetContents*.
- A leitura só ocorre a medida que é necessária, isso permite lidar com arquivos grandes.
- Ao finalizar com o arquivo, é preciso fechar o *handle*.

Usando *withFile*

```haskell
import System.IO

-- withFile :: FilePath -> IOMode ->
                (Handle -> IO a) -> IO a

main = do
       withFile "arquivo_texto.txt" ReadMode
       (\handle -> do
         contents <- hGetContents handle
         putStr contents)
```

Funções para trabalhar com *handles*

- *hGetLine*, *hPutStr*, *hPutStrLn*, *hGetChar*.
- Atuam no *handle* de um arquivo.
- *putStrLn* recebe uma *string* e retorna uma ação de E/S que escreve no terminal.
- *hPutStrLn* recebe um *handle* e uma *string* e retorna uma ação de E/S que escreve a *string* no arquivo associado ao *handle*.

readFile

```haskell
import System.IO
-- readFile :: FilePath -> IO String
main = do
    putStrLn "Informe o caminho de um arquivo:"
    arquivo <- getLine
    contents <- readFile arquivo
    putStr contents
```

writeFile

```haskell
import System.IO
import Data.Char

main = do
       putStrLn "Informe o caminho de um arquivo:"
       arquivo <- getLine
       contents <- readFile arquivo
       writeFile (arquivo ++ ".UPPER")
                 (map toUpper contents)
```

appendFile

```haskell
import System.IO

main = do
        entrada <- getLine
        appendFile "diario.txt" (entrada ++ "\n")
```

Lista de Tarefas

```
import System.IO
import System.Directory
import Data.List

main = do
    handle <- openFile "listadeatividades.txt" ReadMode
    (tempName, tempHandle) <- openTempFile "." "temp"
    contents <- hGetContents handle
    let tarefas = lines contents
        tarefasNumeradas = zipWith (\n line -> show n ++ " - " ++ line)
                                   [0..] tarefas
    putStrLn "Estas são suas tarefas:"
    putStr $ unlines tarefasNumeradas
    putStrLn "Qual você deseja deletar?"
    numeroDaTarefa <- getLine
    let numero = read numeroDaTarefa
        novasTarefas = delete (tarefas !! numero) tarefas
    hPutStr tempHandle $ unlines novasTarefas
    hClose handle
    hClose tempHandle
    removeFile "listadeatividades.txt"
    renameFile tempName "listadeatividades.txt"
```

Argumentos de Linha de Comando

- Nem sempre é uma boa ideia exibir um *prompt* e perguntar ao usuário.
- Muitas vezes, na própria invocação do programa, você quer passar informações para o código.
- Esse cenário é muito útil na execução de programas em servidores.
- Módulo *System.Environment*:
  - *getArgs :: IO [String]*
  - *getProgName :: IO String*

Exemplo de Captura de Argumentos

```
import System.Environment
import Data.List

main = do
       args <- getArgs
       progName <- getProgName
       putStrLn "Os argumentos são:"
       mapM putStrLn args
       putStrLn "O nome do programa é:"
       putStrLn progName
```

Programa para Gerenciar suas Atividades

Vamos fazer um programa para:

- Visualizar as tarefas.
- Adicionar tarefas.
- Deletar tarefas.

Ações

```
import System.Environment
import System.Directory
import System.IO
import Data.List

comandos :: [(String, [String] -> IO ())]
comandos =  [ ("adicionar", adicionar)
            , ("exibir", exibir)
            , ("remover", remover)
            ]
```

Adicionar e Exibir

```
adicionar :: [String] -> IO ()
adicionar [arquivo, atividade] = appendFile
                                 arquivo (atividade ++ "\n")

exibir :: [String] -> IO ()
exibir [arquivo] = do
    contents <- readFile arquivo
    let atividades = lines contents
        atividadesNumeradas = zipWith
    (\n line -> show n ++ " - " ++ line) [0..] atividades
    putStr $ unlines atividadesNumeradas
```

Remover

```
remover :: [String] -> IO ()
remover [arquivo, numeroString] = do
    handle <- openFile arquivo ReadMode
    (tempName, tempHandle) <- openTempFile "." "temp"
    contents <- hGetContents handle
    let numero = read numeroString
        atividades = lines contents
        novasAtividades = delete (atividades !! numero) atividades
    hPutStr tempHandle $ unlines novasAtividades
    hClose handle
    hClose tempHandle
    removeFile arquivo
    renameFile tempName arquivo

main = do
    (comando:args) <- getArgs
    let (Just acao) = lookup comando comandos
    acao args
```

Número Aleatórios

- A maioria das linguagens oferece uma maneira de gerar número *pseudo* aleatórios.
- Em geral, é uma função ou método que toda vez que é invocada, retorna um número diferente, *aleatório*.
- No mundo da pureza funcional, você não pode ter uma função que toda vez que é executada com os mesmos argumentos, retorna valores diferentes.

```
-- sudo apt install libghc-random-dev
import System.Random
random :: (RandomGen g, Random a) => g -> (a, g)
```

Exemplos

```
> import System.Random
> random (mkStdGen 100)
(-3633736515773289454,693699796 2103410263)
> random (mkStdGen 100)
(-3633736515773289454,693699796 2103410263)
> random (mkStdGen 100) :: (Float, StdGen)
(0.6512469,651872571 1655838864)
> random (mkStdGen 100) :: (Bool, StdGen)
(True,4041414 40692)
> random (mkStdGen 101) :: (Float, StdGen)
(0.88034075,105509204 1655838864)
> random (mkStdGen 102) :: (Float, StdGen)
(0.109434664,1706629400 1655838864)
```

Jogando Moedas

```
import System.Random

tresMoedas :: StdGen -> (Bool, Bool, Bool)
tresMoedas gen =
    let (primeiraMoeda, novoGen) = random gen
        (segundaMoeda, novoGen') = random novoGen
        (terceiraMoeda, novoGen'') = random novoGen'
    in  (primeiraMoeda, segundaMoeda, terceiraMoeda)
```

Realizando E/S para obter Gerador

```
import System.Random

main = do
    gen <- getStdGen
    putStr $ take 20 (randomRs ('a','z') gen)
```

Gerando Números Aleatórios

```haskell
import System.Random

main = do
    gen <- getStdGen
    let (n, g) = randomR (1::Int, 10::Int) gen
    putStrLn $ show (n)
```