Programação Funcional e Lógica

Entrada e Saída em Haskell

Prof. Ricardo Couto A. da Rocha
rcarocha@ufg.br
UFG – Regional de Catalão

- *Baseado no livro "Learn Haskell for Great Good". Miran Lipovaca. http://learnyouahaskell.com/*

Leitura de Referência

Aprender Haskell será um grande bem para você - Livro-tutorial online (tradução do livro de Miran Lipovaca)

- **Capítulo 9: Entrada e Saída**

`http://haskell.tailorfontela.com.br/input-and-output`

Entrada e Saída

- Entrada e Saída é uma contradição à Programação Funcional
  - **Objetivo** de projeto de linguagem funcional
    - Espelhar o mais próximo e amplamente possível as funções matemáticas
  - Processo de computação em LF é diferente de linguagens imperativas
    - **Imperativa** → operações são realizadas e valores armazenados em variáveis (seu gerenciamento é fonte de complexidade)
    - **Funcional** → não há variáveis e não há estado da computação
  - **Transparência referencial** → avaliação de função sempre produz o mesmo resultado para mesmos parâmetros
- Programação funcional não deve possuir efeito colateral
  - Entrada e saída geram efeito colateral
- Esses problemas são resolvidos de maneira elegante em Haskell sem comprometer suas características de linguagem funcional.

Operações de Entrada e Saída

- Exemplo de Hello World em Haskell

```
main = putStrLn "hello, world"
```

- Verifique o tipo da função **putStrLn** no GHCi.
  - Use o comando **:t** do interpretador
- IO significa uma ação de entrada e saída.
  - **Ação de IO** → quando executada gera um efeito colateral e terá algum tipo de retorno dentro dele.
- A tupla vazia significa que o tipo retornado na expressão é nulo

Operações de IO

- Entrada e saída ocorrem na função **main**
  - Para mais de uma entrada ou saída devemos usar a sintaxe **do**
- Exemplo

```
main = do
    putStrLn "Qual é o seu nome?"
    name <- getLine
    putStrLn ("Olá " ++ name ++ ", bom dia!")
```

- Todos os passos são ações de I/O
  - Todas as invocações quando avaliadas devem executar ações IO
- **main** segue sempre a declaração **main :: IO alguma-coisa**
- no caso **IO ()** pois é a última ação realizada.
- **name <- getLine**
  - **name** não é uma variável e nem tem significado de atribuição
  - Verifique no GHCi qual o tipo de **getLine**

Associação a variáveis no main

- Considere o seguinte código a ser colocando dentro do main.

```
nameTag = "Hello, my name is " ++ getLine
```

- O trecho de código não é válido!
  - Uma verificação do tipo de **getLine** mostra que ele não é compatível com o que o operador de concatenação está esperando.
  - **nameTag** é uma String e uma operação dentro do **main** deve necessariamente executar uma ação IO.

let em Haskell

- Uma alternativa para nomes a resultados de funções dentro do **main** é o uso de associações **let.**

```
let a = “universidade”
    al = length a
putStr $ a ++ “ tem tamanho “
print al
```

- As associações especificadas no let são avaliadas apenas onde elas são encontradas
  - Apenas nas linhas seguintes é que **length a** é avaliado.

Uso do let no main

- Exemplo de uso do let

```
import Data.Char
main = do
    putStrLn "What's your first name?"
    firstName <- getLine
    putStrLn "What's your last name?"
    lastName <- getLine
    let bigFirstName = map toUpper firstName
        bigLastName = map toUpper lastName
    putStrLn $ "hey " ++ bigFirstName ++ " "
                      ++ bigLastName ++
                         ", how are you?"
```

Exemplo de Programa Interativo

- Objetivo → criar um programa que
  - Continuadamente lê uma linha
  - Imprime o conteúdo da link com cada palavra de trás para frente
  - Programa termina quando uma linha vazia (enter)
- Reflita como seria a estrutura do seu programa em uma linguagem imperativa
- Estrutura do programa em Haskell

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

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

Função return

- Significado de `return ()` :
  - Completamente diferente do que estamos acostumados em linguagens imperativas, quando representa o ponto de retorno de uma função
  - Executa uma ação de IO que não faz nada, mas que resulta no valor indicado.
  - Um return sozinho faz com que o resultado não seja associado com nenhum nome. Por outro lado, `a <- return 22` é uma construção válida.
- Considere o seguinte código

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

- O que esse código fará?

Outras operações de IO

- Observe o cartão de referência
  - Página 2, coluna 3
- Exemplos de operações
  - `putChar`
  - `putStr`
  - `putStrLn`
  - `print`
  - `getChar`
  - `getLine`
  - `getContents`

Exercício

- Faça um programa Haskell que receba Strings digitadas pelo usuário e para cada string **s** escreva:
  - "A string **s** não é palíndromo!", caso a string não seja palíndroma.
  - "A string **s** é palíndromo!", caso a string seja palíndroma.
    - Neste caso, o programa deve terminar.