Linguagens de Programação

Funções e Definições Locais

Samuel da Silva Feitosa

Aula 8

Funções

Definindo Funções

- Além de poder usar funções das bibliotecas, o programador pode definir suas próprias funções.
  - Novas funções são definidas em arquivos de texto geralmente chamados de código-fonte ou script.
  - Um programa fonte contém definições usadas para estruturar o código da aplicação.
  - Por convenção, arquivos fonte em Haskell tem extensão .hs.
- Funções são definidas usando equações.
  - No lado esquerdo colocamos o nome da função e seus parâmetros formais.
  - No lado direito colocamos a expressão da função.

Convenções

- Nomes de funções podem ser alfanuméricos ou simbólicos.
- Identificadores alfanuméricos
  - Começam com letra minúscula e podem conter letras, dígitos, *underline*, e aspas simples.
  - Exemplos: myFun, fun1, arg_2, x’
- Identificadores simbólicos
  - Formados por uma sequência de símbolos.
  - Exemplos: <+>, ===, $*=*$, +=
- Ao desenvolver um programa pode ser útil manter duas janelas: editor de texto e GHCi.

Meu primeiro programa fonte

- Em um editor de texto, vamos digitar as seguintes funções, e salvar como test.hs.

```
-- calcula o dobro de um número
dobro x = x + x

-- calcula o quádruplo de um número
quadruplo x = dobro (dobro x)
```

- No terminal vamos executar o GHCi.

```
$ ghci test.hs
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling Main             ( test.hs, interpreted )
Ok, modules loaded: Main.

*Main>
```

Testando as funções

- Agora, tanto *Prelude.hs* quanto *test.hs* são carregados e podemos usar funções de ambos.

```
*Main> quadruplo 10
40

*Main> 5*(dobro 2) - 3
17
```

- Observe que o GHCi usa o nome de módulo *Main* se o arquivo fonte não define um nome para o módulo.

Modificando o programa

- Com o GHCi aberto, vamos editar o arquivo fonte adicionando uma nova função.

```
areaCirculo r = pi * r^2
```

- O GHCi não detecta mudanças automaticamente
  - O comando *:reload* serve para recarregar o arquivo.

```
*Main> :reload
[1 of 1] Compiling Main             ( test.hs, interpreted )
Ok, modules loaded: Main.
```

```
*Main> areaCirculo 5
78.53981633974483
```

Condicionais com Guardas

- Funções podem ser definidas através de **equações com guardas**.
  - sequência de expressões lógicas usadas para escolher entre vários possíveis resultados.
- Como exemplo, considere uma função para calcular o valor absoluto de um número:

```
vabs :: Integer -> Integer
vabs n | n >= 0 = n
       | n < 0  = - n
```

```
sinal :: Int -> Int
sinal n | n < 0     = -1
        | n == 0    = 0
        | otherwise = 1
```

Definições locais em equações

- Em muitas situações é desejável poder definir valores ou funções auxiliares em uma definição principal.
  - Isso pode ser feito utilizando a cláusula *let* no início da equação ou *where* ao final da equação.
  - A cláusula *let* faz definições locais temporárias, com escopo dentro da função que a define.
  - A cláusula *where* faz definições locais à equação, ou seja, o escopo dos nomes definidos em uma cláusula *where* restringe-se a função que a define.

Exemplo - *let*

- Considere a fórmula: $A = \sqrt{s(s-a)(s-b)(s-c)}$
- Para calcular a área de um triângulo com lados a, b e c, sendo:

$$s = \frac{a+b+c}{2}$$

- Como s aparece várias vezes, podemos defini-lo localmente uma única vez e reutilizá-lo através do comando *let*.

```
areaTriagulo a b c = let s = (a + b + c) / 2
                     in sqrt (s * (s-a) * (s-b) * (s-c))
```

Exemplo - *where*

- Considere a fórmula: $A = \sqrt{s(s-a)(s-b)(s-c)}$
- Para calcular a área de um triângulo com lados a, b e c, sendo:

$$s = \frac{a+b+c}{2}$$

- Como s aparece várias vezes, podemos defini-lo localmente uma única vez e reutilizá-lo através do comando *where*.

```
areaTriangulo a b c = sqrt (s * (s-a) * (s-b) * (s-c))
  where
    s = (a + b + c)/2
```

Execução do exemplo - *where*

- Esta definição assume que os argumentos da função são valores válidos para os lados de um triângulo.

```
areaTriangulo 5 6 8
⇝ sqrt (s * (s-5) * (s-6) * (s-8))
   where
     s = (5 + 6 + 8)/2
         ⇝ 9.5
⇝ sqrt (9.5 * (9.5-5) * (9.5-6) * (9.5-8))
⇝ sqrt 224.4375
⇝ 14.981238266578634
```

Diferenças entre *let* e *where*

- Com **where** as definições são colocadas no final, e com **let** elas são colocadas no início.
- O **let** é uma expressão e pode ser usada em qualquer lugar onde se espera uma expressão.
- Já **where** não é uma expressão, podendo ser usada apenas para fazer definições locais em uma definição de função.

Definições locais em equações

- Tanto funções como ligações temporárias podem ser definidas localmente.
  - A ordem das equações locais é irrelevante.

```
minhaFuncao x = 3 + f x + f a + f b
  where
    f x = x + 7*c
    a = 3*c
    b = f 2
    c = 10
```

Execução de minhaFuncao

```
minhaFuncao 5
⇝ 3 + f 5 + f a + f b
   where f x = x + 7*c
         a = 3*c
             ⇝ 3*10
             ⇝ 30
         b = f 2
             ⇝ 2 + 7*10
             ⇝ 2 + 70
             ⇝ 72
         c = 10
⇝ 3 + (5 + 7*10) + (30 + 7*10) + (72 + 7*10)
⇝ 3 + (5 + 70) + (30 + 70) + (72 + 70)
⇝ 3 + 75 + 100 + 142
⇝ 320
```

Funções com Listas

- Dada uma lista de strings, a função deve retornar a primeira string da lista ou a palavra “vazia”, caso a lista não tenha elementos.

```
firstOrEmpty lst = if not (null lst)
                   then head lst
                   else "empty"
```

- Testando a função:

```
*Main> firstOrEmpty []
"empty"
*Main> firstOrEmpty ["Hello", "Ola"]
"Hello"
```

Especificando o tipo da função

- Como vimos, Haskell é uma linguagem estaticamente e fortemente tipada.
  - Quando não especificamos o tipo, o sistema utiliza um mecanismo de inferência de tipos.
  - Entretanto, não é uma boa prática deixar funções sem especificar os tipos.
- Vamos especificar o tipo da função *firstOrEmpty*.

```
firstOrEmpty :: [[Char]] -> [Char]
firstOrEmpty lst = if not (null lst)
                   then head lst
                   else "empty"
```

Lidando com Entrada e Saída

- Em linguagens puras o valor retornado por uma função **depende única e exclusivamente** dos argumentos especificados na aplicação da função.
  - Assim não é possível implementar uma função que lê um caracter da mesma maneira que em linguagens impuras.
- Para interagir com o usuário, precisamos de uma representação do sistema de computação onde o programa está sendo executado.
  - O **mundo** é formado por todas as informações no contexto de execução da aplicação: entrada e saída padrão, discos, rede, etc.

Entrada e Saída - Exemplos

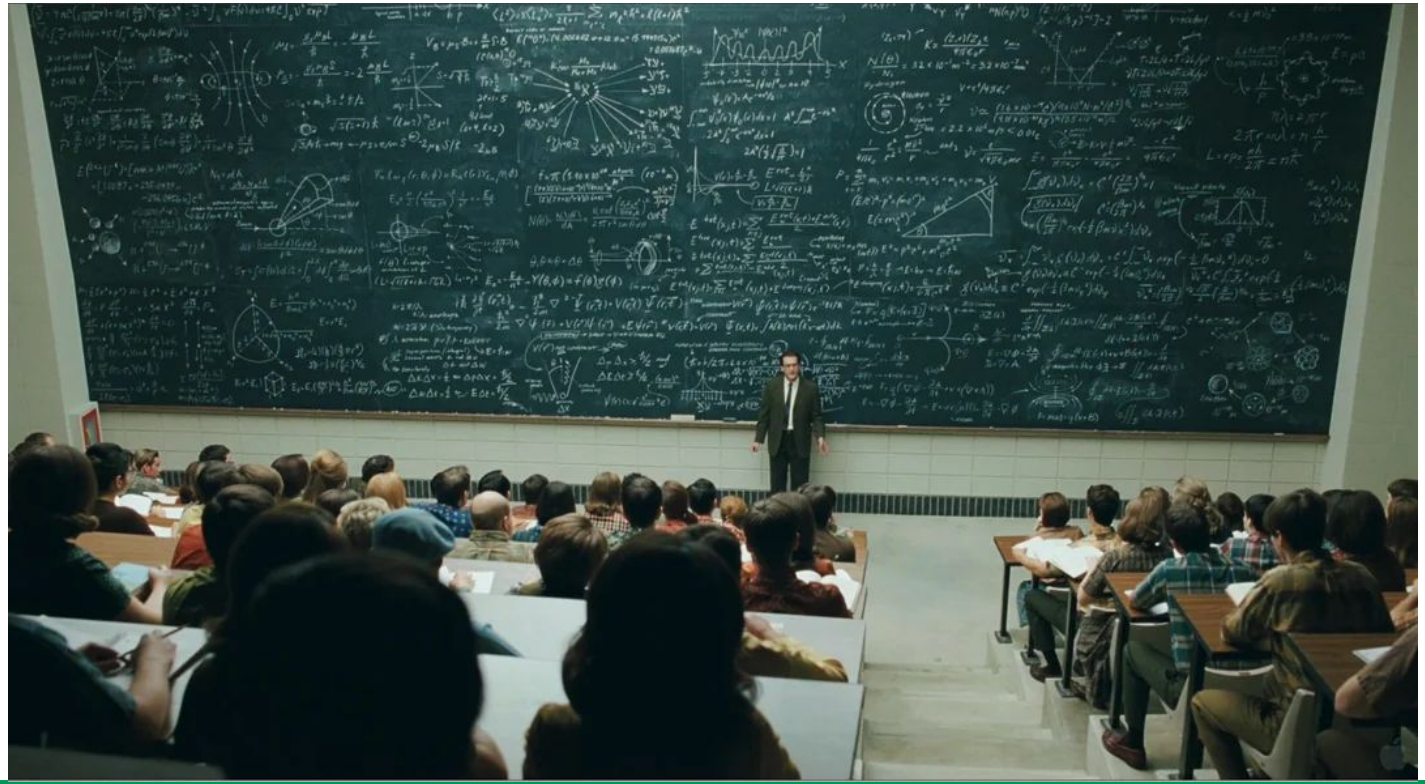
```haskell
module Main (main) where

main :: IO ()
main = do putStrLn "Qual é o seu nome? "
          nome <- getLine
          putStr nome
          putStrLn ", seja bem vindo(a)!"
```

```haskell
module Main (main) where

main :: IO ()
main = do putStrLn "Digite um número: "
          n1 <- readLn
          putStrLn "Digite outro número: "
          n2 <- readLn
          putStr "Soma dos números digitados: "
          putStrLn (show (n1 + n2))
```

E com isso fazemos um Hello World em Haskell!



Considerações Finais

- Aprendemos como criar funções em Haskell.
  - Devemos seguir as convenções na criação de funções.
- Escrevemos nosso primeiro código-fonte e testamos o código via GHCi.
- Trabalhamos com definições locais.
  - Utilizamos as palavras-chave *let* e *where*.
- Utilizamos uma função para lidar com Listas.
  - E definimos o tipo da função estaticamente.

Exercícios de Fixação

1. Defina uma função para calcular o quadrado do dobro do seu argumento.
2. Defina uma função para calcular o dobro do quadrado do seu argumento.
3. Faça uma função que receba três notas de um aluno e calcule a média aritmética das notas.
4. Sabe-se que o valor do quilowatt de energia elétrica custa um quinto do salário mínimo. Defina uma função que receba o valor do salário mínimo e a quantidade de quilowatts consumida por uma residência, e resulta no valor a ser pago com desconto de 15%.