

# Programação Funcional

## Funções e Tipos de Dados de Usuário

**Prof. Edson Alves**

Faculdade UnB Gama

# Sumário

1. Funções
2. Tipos de dados de usuário

## Aplicação de funções

- ▶ Haskell não utiliza parêntesis para delimitar os parâmetros de uma função em uma chamada
- ▶ A sintaxe para a chamada de funções é

```
nome_da_funcao param1 param2 ... paramN
```

- ▶ Por exemplo, o código abaixo compara dois inteiros:

```
ghci> compare 5 3
```

- ▶ Os parêntesis são utilizados para resolver ambiguidades ou clarificar o significado de expressões complexas
- ▶ Por exemplo, o código abaixo compara as raízes quadradas de dois inteiros

```
ghci> compare (sqrt 5) (sqrt 3)
```

## Exemplos de funções que agem em listas

1. A função **head** retorna o primeiro elemento de uma lista. Ex.:

```
ghci> head [2..5]           -- retorna 2
```

2. Já a função **tail** retorna todos os elementos da lista, exceto o primeiro. Ex.:

```
ghci> tail [2..5]           -- retorna [3, 4, 5]
```

3. A função **take** extrai os  $n$  primeiros elementos da lista. Ex.:

```
ghci> take 5 "Hello World!" -- retorna "Hello"
```

4. A função **drop** retorna todos os elementos da lista, exceto os  $n$  primeiros. Ex.:

```
ghci> drop 6 "Hello World!" -- retorna "World!"
```

5. A função **length** retorna o número de elementos da lista

```
ghci> length [53..278]      -- retorna 226
```

6. A função **null** retorna verdadeiro se a lista está vazia; e falso, caso contrário

# Tuplas

- ▶ Uma tupla é uma coleção, de tamanho fixo, de objetos de quaisquer tipos
- ▶ Em Haskell, as tuplas são delimitadas por parêntesis
- ▶ Por exemplo, a tupla `(1, True, "Test")` tem tipo `(Int, Bool, [Char])`
- ▶ Uma tupla que não contém nenhum elemento é grafada como `()`
- ▶ Não há tuplas de um único elemento em Haskell
- ▶ A ordem e o tipo dos elementos da tupla fazem diferença
- ▶ Por exemplo, as tuplas `(True, 'a')` e `('a', True)` tem tipos distintos, e não são comparáveis
- ▶ As tuplas podem ser utilizadas para retornar múltiplos valores de uma função
- ▶ As funções `fst` e `snd` retornam o primeiro e segundo elemento de uma tupla de dois elementos, respectivamente

## Definindo novas funções

- ▶ A sintaxe para a definição de uma nova função é

```
nome_da_funcao param1 param2 ... paramN = expressao
```

- ▶ O nome da função deve iniciar em minúsculas e os parâmetros devem ser separados por um espaço em branco
- ▶ O valor da expressão será o retorno da função
- ▶ A declaração do tipo da função é opcional

```
areaCircle :: Double -> Double  
areaCircle r = pi * r ^ 2
```

- ▶ A declaração de tipo é útil para definir o tipo da função em caso de ambiguidade
- ▶ O *currying* é aplicado em funções de dois ou mais parâmetros

```
-- Os parêntesis na declaração de tipo são opcionais  
areaTriangle :: Double -> (Double -> Double)  
areaTriangle b h = (b + h)/2.0
```

# Condicionais

- ▶ Haskell tem uma variante do construto IF-THEN-ELSE
- ▶ A sintaxe é

```
if condicao then valor_se_verdadeiro else valor_se_falso
```

- ▶ A condicao deve ser uma expressão do tipo **Bool**
- ▶ Os dois valores devem ter o mesmo tipo
- ▶ Ao contrário das linguagens imperativas, a cláusula **else** é obrigatória
- ▶ Se indentação for utilizada, ela deve ser consistente, pois fará parte do construto
- ▶ Diferentemente das linguagens imperativas, o uso deste construto não é muito frequentes, sendo preterido nos casos que o *pattern matching* puder ser utilizado

## Exemplo de condicionais em Haskell

```
1 -- Este arquivo pode ser importado no GHCi com o comando :load
2 --
3 --      ghci> :load collatz.hs
4 --
5 -- Após a importação a função collatz estará disponível para uso
6
7
8 -- Sequência de Collatz:  $c(1) = 1$ ,  $c(n) = n/2$ , se  $n$  é par,
9 --  $c(n) = 3*n + 1$ , se  $n$  é ímpar
10 collatz n = n : if n == 1
11                then []
12                else if even n
13                      then collatz (div n 2)
14                      else collatz (3*n + 1)
```



## Lazy evaluation

- ▶ Em Haskell a valoração das expressões é não-estrita (*lazy evaluation*)
- ▶ Isto significa que os termos de uma expressão ou os parâmetros de uma função só serão computados caso sejam necessários
- ▶ As vantagens desta abordagem é que não são feitos cálculos desnecessários
- ▶ Porém é preciso um maior tempo de processamento e memória, pois é preciso manter o registro das expressões intermediárias (*thunks*)
- ▶ Esta estratégia está embutida na linguagem, sem a necessidade de nenhum indicativo ou marcação nos programas
- ▶ A título de exemplo, e para entender as diferenças entre as duas abordagens, a expressão

```
(sum [1..10] > 50) || (length [1..] > 1000)
```

será computada usando as valorações estrita e não-estrita

## Exemplo de valoração

### Valoração estrita:

```
(sum [1..10] > 50) || (length [1..] > 1000)
(sum ([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]) > 50) || (length [1..] > 1000)
(55 > 50) || (length [1..] > 1000)
True || (length [1..] > 1000)
True || (length [1, 2, 3, ...] > 1000)      -- Laço infinito
```

### Valoração não-estrita:

```
(sum [1..10] > 50) || (length [1..] > 1000)
(sum ([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]) > 50) || (length [1..] > 1000)
(55 > 50) || (length [1..] > 1000)
True || (length [1..] > 1000)
True
```

# Polimorfismo

- ▶ Em Haskell, listas são tipos polimórficos
- ▶ A notação `[a]` significa “uma lista de elementos do tipo `a`”
- ▶ Variáveis que correspondem a tipos de dados começam sempre em minúsculas
- ▶ Não há maneiras de se determinar exatamente qual é o tipo de `a`
- ▶ A função `fst` tem o tipo `fst :: (a, b) -> a`
- ▶ `a` e `b` representam tipos, possivelmente distintos
- ▶ Pelo tipo da função, o único comportamento possível que ela pode ter é retornar o primeiro elemento da tupla
- ▶ Esta é uma importante característica do Haskell: o tipo de uma função pode dar pistas sobre (ou possivelmente determinar) o comportamento de uma função

## Funções puras

- ▶ Haskell assume, por padrão, que todas as funções são puras
- ▶ Esta características tem profundas implicações na linguagem e na forma de programar
- ▶ Por exemplo, considere que a função `f` tenha o tipo `f :: Bool -> Bool`
- ▶ Como `f` é pura, ela só pode ter um dos seguintes comportamentos:
  - i. ignorar seu argumento e retornar sempre `True` ou `False`
  - ii. retornar seu argumento sem modificações
  - iii. negar seu argumento
- ▶ A pureza das funções é inerentemente modular: toda função é auto-contida e tem uma interface bem definida
- ▶ Ela também facilita o teste unitário de cada função
- ▶ Em Haskell, códigos impuros devem ser separados de códigos puros, e a maior parte do programa deve ser puro: a parte impura deve ser a mais simples possível

## Definição de novos tipos de dados

- ▶ É possível introduzir novos tipos de dados por meio da palavra reservada **data**
- ▶ A sintaxe é

```
data construtor_do_tipo = construtor_de_valor tipo1 tipo2 ... tipoN
```

- ▶ Tanto o construtor de tipo quanto o construtor de valor devem iniciar em letra maiúscula
- ▶ Os  $N$  tipos se referem aos tipos dos  $N$  membros (campos) do novo tipo de dado
- ▶ O construtor de valor pode ser entendido como uma função qualquer
- ▶ Por exemplo,

```
-- Definição do novo tipo
data StudentInfo = Student String Int Double deriving (Show)

-- Nova variável do tipo recém-criado
newStudent = Student "Fulano de Tal" 20 1.77
```

## Definição de novos tipos de dados

- ▶ O nome do tipo e de seus valores são independentes
- ▶ Os nomes dos tipos são usados exclusivamente em suas definições
- ▶ Os construtores de valores são utilizados no programa para criar variáveis do tipo definido
- ▶ Quando não há ambiguidade, os nomes dos tipos e dos valores podem ser iguais
- ▶ Esta prática é normal e legal
- ▶ Haskell não permite a mistura de dois tipos de dados que são estruturalmente iguais, mas tem nomes diferentes
- ▶ Por exemplo, no trecho abaixo,

```
data Point2D = Point2D Double Double
data Polar = Polar Double Double
```

```
x = Point2D 1.0 2.0
y = Polar 1.0 2.0
```

x e y não são comparáveis, pois tem tipos distintos

# Sinônimos

- ▶ Em Haskell é possível definir um **sinônimo** para um tipo já existe, com o intuito de ampliar o entendimento do código por meio do uso de nomes mais descritivos
- ▶ Isto pode ser feito por meio da palavra-reservada **type**
- ▶ Por exemplo,

```
-- Sinônimos para os campos
type Name = String
type UID = Int
type Score = Double

-- Mesmo tipo anterior, porém com nomes de campos mais descritivos
data Student = Student Name UID Score deriving (Show)
```

- ▶ A definição de sinônimos é similar a feita em C/C++ por meio da palavra-reservada **typedef**

## Tipos de dados algébricos

- ▶ O tipo de dados **algébricos** podem ter mais de um construtor. Por exemplo:

```
-- Eq permite que tipos sejam comparáveis por igualdade  
data BloodGroup = A | B | AB | O deriving (Eq, Show)
```

- ▶ Eles também não denominados tipos **enumeráveis**
- ▶ Cada um dos construtores pode ter zero ou mais parâmetros. Por exemplo:

```
type Radius = Double  
type Base = Double  
type Height = Double  
  
data Shape = Circle Radius  
           | Triangle Base Height deriving (Eq, Show)
```

- ▶ Com um único construtor, o tipo de dado algébrico equivale a uma **struct** em C
- ▶ Com dois ou mais construtores, todos sem parâmetros, corresponde a uma **enum**
- ▶ Nos demais casos, pode ser visto como uma **union** de C/C++



## Pattern Matching

- ▶ Um padrão (*pattern*) permite a extração de membros de um tipo
- ▶ Isto permite a definição de uma função por meio de vários padrões de entrada
- ▶ Quando a função for chamada, os valores dos parâmetros de entrada são confrontados com os padrões definidos, um por vez, na ordem em que foram definidos
- ▶ Quando um casamento (*matching*) for bem sucedido, a expressão associada definirá o valor de retorno da função
- ▶ Se nenhum casamento for bem sucedido, o resultado será um erro
- ▶ O símbolo '\_' (*wildcard*) pode ser usado para casar qualquer valor
- ▶ Em alguns contextos, o *pattern matching* é também denominado **desconstrução**

## Exemplo de definição de função usando *pattern matching*

```
1 type Radius = Double
2 type Base = Double
3 type Height = Double
4
5 data Shape = Circle Radius
6             | Triangle Base Height deriving (Eq, Show)
7
8 -- A função area() é definida para diferentes padrões de entrada
9 area (Circle r)      = pi * r ^ 2
10 area (Triangle b h) = (b * h) / 2
11
12 main = print (area x, area y) where
13     x = Circle 2.0
14     y = Triangle 3.0 4.0
```

## Exemplo de *pattern matching* em listas

```
1 -- Verifica se a lista está vazia
2 sumList [] = 0
3
4 -- Verifica se a lista é composta pela concatenação do elemento x
5 -- com a sublista xs. Usar uma letra seguida de um s é uma convenção
6 -- de nomenclatura para listas, onde xs significa: 'uma lista de
7 -- (vários) elementos x', com sentido de plural
8 sumList (x:xs) = x + sumList xs
9
10 main = print p where
11     p = sumList [1..100]
```

## Exemplo de extração de membros usando *pattern matching*

```
1 type Name = String
2 type UID = Int
3 type Score = Double
4
5 data Student = Student Name UID Score deriving (Show)
6
7 -- O caractere _ é o wildcard e casa com qualquer padrão
8 studentName (Student name _ _) = name
9 studentUID (Student _ uid _) = uid
10 studentScore (Student _ _ score) = score
11
12 -- Mais sobre o caractere $ no próximo slide
13 main = putStrLn $ "Nome: " ++ studentName s where
14     s = Student "Fulano de Tal" 12345 8.7
```

## O símbolo \$

- ▶ O símbolo \$ é um operador em Haskell com um comportamento que não é óbvio a primeira vista
- ▶ O tipo do operador \$ é

```
($) :: (a -> b) -> a -> b
```

- ▶ Ele recebe uma função `f :: a -> b` e o parâmetro `a`, e retorna `b = f(a)`
- ▶ Este é o mesmo comportamento do espaço em branco:

```
f :: a -> b  
x :: a  
f $ x = f x
```

- ▶ A importância deste operador vem de sua precedência

```
ghci> :info ($)  
($) :: (a -> b) -> a -> b  
      -- Defined in 'GHC.Base'  
infixr 0 $
```

## O símbolo \$

- ▶ **infixr** nos diz que este é um operador associativo à direita
- ▶ O valor 0 (zero) corresponde a menor precedência possível
- ▶ Já a aplicação de função (operador espaço) é associativo à esquerda e tem a maior precedência possível
- ▶ Assim, ele serve para mudar a associatividade e precedência da aplicação de funções
- ▶ Por exemplo, a expressão

```
sqrt head [2..5]
```

é inválida, pois a associatividade e a precedência da aplicação faz com que ela seja interpretada como

```
(sqrt head) [2..5]
```

- ▶ Já com o uso do operador dólar temos

```
sqrt $ head [2..5] = sqrt (head [2..5]) = sqrt 2 = 1.41421356...
```

## Notação de registro

- ▶ Escrever funções de acesso para os membros de um tipo de dado é tedioso e propenso a erros
- ▶ Haskell oferece uma definição alternativa, que permite listar as funções de acesso no momento da definição do novo tipo de dado
- ▶ O tipo **Student** poderia ser definido da seguinte forma:

```
type Name = String
type UID = Int
type Score = Double

data Student = Student {
    studentName  :: Name,
    studentUID   :: UID,
    studentScore :: Score
} deriving (Show)
```

- ▶ Esta notação também pode ser utilizada para criar variáveis deste novo tipo
- ▶ A impressão padrão herdada de **Show** também fica mais elaborada

## Exemplo de criação de variável com notação de registro

```
1 type Name = String
2 type UID = Int
3 type Score = Double
4
5 data Student = Student {
6     studentName  :: Name,
7     studentUID   :: UID,
8     studentScore :: Score
9 } deriving (Show)
10
11 -- Definição de variável usando notação de registro
12 s = Student {
13     studentName = "Fulano de Tal",
14     studentUID  = 12345,
15     studentScore = 8.7
16 }
17
18 main = print $ studentScore s
```



## Tipos parametrizáveis

- ▶ Em Haskell é possível criar tipos de dados parametrizados, por meio da introdução de uma variável na definição de tipo
- ▶ Na biblioteca padrão (**Prelude**) é definido o tipo parametrizado **Maybe**:

```
data Maybe a = Just a | Nothing
```

- ▶ A variável **a** nesta definição significa um tipo de dado qualquer
- ▶ Este tipo de dado é utilizado para retornos de funções que podem estar ausentes
- ▶ Por exemplo, a função **head** retorna um erro quando aplicada à lista vazia

```
ghci> head []  
*** Exception: Prelude.head: empty list
```

- ▶ Uma versão “segura” de **head** pode ser implementada da seguinte maneira:

```
safeHead [] = Nothing  
safeHead (x:xs) = Just x
```

# Tipos recursivos

- ▶ Um tipo é dito **recursivo** se ele for definido em termos de si próprio
- ▶ Por exemplo, uma lista parametrizada pode ser definida como

```
data List a = Cons a (List a)
            | Null
            deriving (Show)
```

- ▶ Os tipos recursivos devem ter ao menos um caso base (construtor que não se refere ao próprio tipo)
- ▶ Assim, eles devem ter, no mínimo, dois construtores distintos
- ▶ Uma árvore binária pode ser definida como

```
data Tree a = Node a (Tree a) (Tree a)
            | Empty
            deriving (Show)
```

## Exemplo de definição e uso de tipo recursivo

```
1 -- Definição alternativa de lista
2 data List a = Cons a (List a)
3             | Null
4             deriving (Show)
5
6 -- Define uma lista do novo tipo
7 xs = Cons 1 (Cons 2 (Cons 3 Null))
8
9 -- Converte da lista definida para a lista padrão do Haskell
10 toList Null = []
11 toList (Cons x xs) = x : toList xs
12
13 main = print (xs, toList xs)
```

## Variáveis locais

- ▶ Em Haskell há duas formas de introduzir variáveis locais a uma função
- ▶ A primeira delas é utilizando uma expressão **let**, cuja sintaxe é

```
nome_da_funcao par1 par2 ... parN = let var1 = value1
                                     var2 = value2
                                     ...
                                     varM = valueM
                                     in expressao_da_funcao
```

- ▶ Cada linha define uma nova variável
- ▶ O escopo destas variáveis é a expressão que define o valor da função e também as linhas de definição de variáveis subsequentes
- ▶ Se o nome de uma das variáveis locais coincidir com o nome de um dos parâmetros da função, prevalecerá a variável local (*shadowing*), o que pode levar a expressões confusas ou *bugs*

## Variáveis locais

- ▶ A segunda forma de se declarar variáveis locais é por meio de uma expressão **where**, cuja sintaxe é

```
nome_da_funcao par1 par2 ... parN = expressao_da_funcao
  where var1 = value1
        var2 = value2
        ...
        varM = valueM
```

- ▶ A diferença entre as expressões **where** e **let** é que a primeira prioriza a expressão, deixando os detalhes para depois
- ▶ O escopo das variáveis definidas na expressão **where** é a expressão da função, que precede o bloco, e o próprio bloco **where**
- ▶ Além de variáveis locais, é possível definir funções locais nas expressões **let** e **where**
- ▶ A escolha entre estas duas alternativas é uma questão de estilo

## Exemplo de uso de variáveis locais

```
1 --  $p(x) = c$ 
2 roots 0 0 c = []
3
4 --  $p(x) = bx + c$ 
5 roots 0 b c = [-c/b]
6
7 --  $p(x) = ax^2 + bx + c$ 
8 roots a b c = if delta >= 0
9               then [(-b + sqrt delta)/(2*a), (-b - sqrt delta)/(2*a)]
10              else []
11              where delta = b^2 - 4*a*c
12
13 main = print $ roots 1 (-5) 6
```

## Indentação

- ▶ Em Haskell, a indentação e espaços em branco são utilizados para delimitar os blocos
- ▶ A primeira expressão de um programa pode começar em qualquer coluna do texto, desde que as demais expressões de mesmo nível comecem nesta mesma coluna
- ▶ Se a expressão é seguida de uma linha em branco ou por uma expressão em uma coluna mais à direita, ela é considerada uma continuação da expressão anterior
- ▶ Após um **let** ou um **where**, o compilador memorizará a posição do próximo *token*: expressões nas próximas linhas que comecem na mesma posição serão consideradas novas entradas destes blocos
- ▶ Por conta destas regras, é indicado o uso de espaços, e não de tabulações, na indentação do código
- ▶ Embora não seja comum, os blocos podem ser delimitados por chaves ('{' e '}'): neste caso, as regras acima podem ser violadas, usando-se ponto-e-vírgula para separar as expressões

## Expressões *case*

- ▶ As expressões **case** permitem confrontar uma expressão com vários padrões
- ▶ A sintaxe é

```
case expressao of
  padrao1 -> valor1
  padrao2 -> valor2
  ...
  padraoN -> valorN
```

- ▶ O valor da expressão é confrontado com cada um dos padrões, na ordem descrita
- ▶ Caso um casamento seja válido, o valor da expressão **case** será o valor indicado após a seta (->)
- ▶ Também é possível usar o *wildcard*



## Exemplo de expressão `case`

```
1 -- Esta função está disponível na biblioteca Maybe.
2 -- Para utilizá-la sem precisar da implementação abaixo,
3 -- importe esta biblioteca com o comando
4 --
5 -- import Maybe
6 --
7 --
8 -- Ela extrai o valor contido no tipo Maybe (wrapped),
9 -- ou retorna defval, em caso Nothing
10 fromMaybe defval wrapped =
11     case wrapped of
12         Nothing    -> defval
13         Just value -> value
```

# Guardas

- ▶ O uso de *pattern matching* é feito apenas com valores dos objetos
- ▶ Para testes mais elaborados, com condicionais, Haskell disponibiliza as **guardas**
- ▶ Cada padrão a ser casado pode ser seguido por uma ou mais guardas, sendo elas expressões do tipo **Bool**
- ▶ Cada guarda é introduzida pelo símbolo `|` e seguida de um `=` (ou `->`, no caso de expressões **case**)
- ▶ Após o símbolo, segue a expressão a ser utilizada caso o padrão seja casado e a guarda verdadeira
- ▶ A expressão **otherwise** é atada ao valor **True**, de modo que é uma guarda sempre verdadeira, o que melhora a legibilidade do código

## Exemplo de uso de guardas

```
1 -- p(x) = c
2 roots 0 0 c = []
3
4 -- p(x) = bx + c
5 roots 0 b c = [-c/b]
6
7 -- p(x) = ax^2 + bx + c
8 roots a b c | D == 0 = [-b/(2*a)]
9             | D > 0  = [(-b + s)/(2*a), (-b - s)/(2*a)]
10            | otherwise = []
11      where D = b^2 - 4*a*c
12            s = sqrt D
13
14 main = print $ roots 1 (-5) 6
```

## Referências

1. **SHALOM**, Elad. *A Review of Programming Paradigms Throughtout the History – With a Suggestion Toward a Future Approach*, Amazon, 2019.
2. **SULLIVAN**, Bryan O.; **GOERZEN**, John; **STEWART**, Don. *Real World Haskell*, O'Reilly.
3. Type Classes. [dolar sign](#), acesso em 01/03/2020.