Programação Funcional

Funções

Prof. Edson Alves

Faculdade UnB Gama

2020

Sumário

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 raizes 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. De modo similar, 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

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

```haskell
-- Este arquivo pode ser importado no GHCi com o comando :load
--
--      ghci> :load collatz.hs
--
-- Após a importação a função collatz estará disponível para uso


-- Sequência de Collatz: c(1) = 1, c(n) = n/2, se n é par,
-- c(n) = 3*n + 1, se n é ímpar
collatz n = n : if n == 1
                then []
                else if even n
                     then collatz (div n 2)
                     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)
(sum 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)
(sum 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 comportamente 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 inerentemete 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, com a parte impura a mais simples o 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 o mesmo
- Esta prática é normal e legal
- Haskell não permite a mistura de dois tipos de dados que são estruturalmente diferentes, mas tem nomes diferentes
- Por exemplo, no trecho abaixo,

```haskell
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

Referências

1. **SHALOM**, Elad. *A Review of Programming Paradigms Througout the History – With a Suggestion Toward a Future Approach*, Amazon, 2019.
2. **SULLIVAN**, Bryan O.; **GOERZEN**, John; **STEWART**, Don. *Real World Haskell*, O'Relly.