Aula prática 6

Tuplas, Listas, e Polimorfismo Paramétrico

Resumo

Nesta aluna vamos conhecer os tipos tuplas e listas, que são tipos polimórficos pré-definidos em Haskell. Vamos aprender também sobre funções polimórficas.

Sumário

1 Tuplas

Tupla é uma estrutura de dados formada por uma sequência de valores **possivelmente de tipos diferentes**. Os componentes de uma tupla são identificados pela **posição** em que ocorrem na tupla.

Em Haskell uma **expressão tupla** é formada por uma sequência de expressões separadas por vírgula e delimitada por parênteses:

$$(\ exp_1 \ , \ \ldots \ , \ exp_n \ )$$

onde $n >= 0$ e $n \neq 1$, e $exp_1, \ldots, exp_n$ são expressões cujos valores são os componentes da tupla.

De maneira similar, um **tipo tupla** é formado por uma sequência de tipos separados por vírgula e delimitada por parênteses:

$$(\ t_1 \ , \ \ldots \ , \ t_n \ )$$

onde $n >= 0$ e $n \neq 1$, e $t_1, \ldots, t_n$ são os tipos dos respectivos componentes da tupla. Observe que o tamanho de uma tupla (quantidade de componentes) é codificado no seu tipo.

`()` é a **tupla vazia**, do tipo `()`.

Não existe tupla de um único componente.

A tabela a seguir mostra alguns exemplos de tuplas:

tupla	tipo
`('A','t')`	`(Char,Char)`
`('A','t','o')`	`(Char,Char,Char)`
`('A',True)`	`(Char,Bool)`
`("Joel",'M',True,"COM")`	`(String,Char,Bool,String)`
`(True,("Ana",'f'),43)`	`Num a => (Bool,(String,Char),a)`
`()`	`()`
`("nao eh tupla")`	`String`

Vejamos algumas **operações com tuplas** definidas no prelúdio:

- `fst`: seleciona o primeiro componente de um *par*:

```
Prelude> fst ("pedro",19)
"pedro"
```

- `snd`: seleciona o segundo componente de um *par*:

```
Prelude> snd ("pedro",19)
19
```

2 Listas

Lista é uma estrutura de dados formada por uma sequência de valores (elementos) **do mesmo tipo**. Os elementos de uma lista são identificados pela **posição** em que ocorrem na lista.

Em Haskell uma **expressão lista** é formada por uma sequência de expressões separadas por vírgula e delimitada por colchetes:

$$[\ exp_1\ ,\ \ldots\ ,\ exp_n\ ]$$

onde $n >= 0$, e $exp_1, \ldots, exp_n$ são expressões cujos valores são os elementos da lista.

Um **tipo lista** é formado pelo tipo dos seus elementos delimitado por colchetes:

$$[\ t\ ]$$

onde t é o tipo dos elementos da lista. Observe que o tamanho de uma lista (quantidade de elementos) não é codificado no seu tipo.

A tabela a seguir mostra alguns exemplos de listas:

lista	tipo
`['O','B','A']`	`[Char]`
`['B','A','N','A','N','A']`	`[Char]`
`[False,True,True]`	`[Bool]`
`[ [False,True], [], [True,False,True] ]`	`[[Bool]]`
`[1,8,6,10.48,-5]`	`Fractional a => [a]`

Vejamos algumas **operações com listas** definidas no prelúdio:

- `null`: verifica se uma lista é vazia:

```
Prelude> null []
True
Prelude> null [1,2,3,4,5]
False
```

- `head`: seleciona a **cabeça** (primeiro elemento) de uma lista:

```
Prelude> head [1,2,3,4,5]
1
Prelude> head []
*** Exception: Prelude.head: empty list
```

- `tail`: seleciona a **cauda** da lista, ou seja, a lista formada por todos os elementos exceto o primeiro:

```
Prelude> tail [1,2,3,4,5]
[2,3,4,5]
Prelude> tail [5*4, 6*5]
[30]
Prelude> tail [8-1]
[]
Prelude> tail []
*** Exception: Prelude.tail: empty list
```

- `lenght`: calcula o tamanho (quantidade de elementos) de uma lista:

```
Prelude> length [1,2,3,4,5]
5
Prelude> length []
0
```

- `(!!)`: seleciona o i-ésimo elemento de uma lista ($0 \leq i < n$, onde n é o comprimento da lista):

```
Prelude> [1,2,3,4,5] !! 2
3
Prelude> [1,2,3,4,5] !! 0
1
Prelude> [1,2,3,4,5] !! 10
*** Exception: Prelude.(!!): index too large
Prelude> [1,2,3,4,5] !! (-4)
*** Exception: Prelude.(!!): negative index
```

- `take`: seleciona os primeiros n elementos de uma lista:

```
Prelude> take 3 [1,2,3,4,5]
[1,2,3]
```

- `drop`: remove os primeiros n elementos de uma lista:

```
Prelude> drop 3 [1,2,3,4,5]
[4,5]
```

- `sum`: calcula a soma dos elementos de uma lista de números:

```
Prelude> sum [1,2,3,4,5]
15
```

- `product`: calcula o produto dos elementos de uma lista de números:

```
Prelude> product [1,2,3,4,5]
120
```

- `(++)`: concatena duas listas:

```
Prelude> [1,2,3] ++ [4,5]
[1,2,3,4,5]
```

- `reverse`: inverte uma lista:

```
Prelude> reverse [1,2,3,4,5]
[5,4,3,2,1]
```

- `zip`: junta duas listas em uma única lista formada pelos pares dos elementos correspondentes:

```
Prelude> zip ["pedro","ana","carlos"] [19,17,22]
[("pedro",19),("ana",17),("carlos",22)]
```

3 Strings

Em Haskell **strings** são **listas de caracteres**. O tipo `String` é um sinônimo para o tipo `[Char]`. A tabela a seguir mostra alguns exemplos de strings:

string	notação de lista
`"ufop"`	`['u','f','o','p']`
`"bom\ndia"`	`['b','o','m','\n','d','i','a']`
`""`	`[]`

4 Polimorfismo paramétrico

4.1 Operação sobre vários tipos de dados

Algumas funções podem operar sobre vários tipos de dados. Por exemplo: a função `head` recebe uma lista e retorna o primeiro elemento da lista:

```
head ['b','a','n','a','n','a']    ⇝ 'b'
head ["maria","paula","peixoto"] ⇝ "maria"
head [True,False,True,True]      ⇝ True
head [("ana",2.8),("pedro",4.3)] ⇝ ("ana",2.8)
```

Não importa qual é o tipo dos elementos da lista. Qual deve ser o tipo de `head`?

```
head :: [Char] -> Char
head :: [String] -> String
head :: [Bool] -> Bool
head :: [(String,Double)] -> (String,Double)
```

`head` pode ter vários tipos.

4.2 Variáveis de tipo

Quando um tipo pode ser *qualquer* tipo da linguagem, ele é representado por uma **variável de tipo**.

No exemplo dado, sendo `a` o tipo dos elementos da lista que é passada como argumento para a função `head`, então

```
head :: [a] -> a
```

`a` é uma *variável de tipo* e pode ser substituída por qualquer tipo. O tipo de `head` estabelece que `head` recebe uma lista com elementos de um tipo qualquer, e retorna um valor deste mesmo tipo.

Em Haskell variáveis de tipo devem começar com uma *letra minúscula*, e são geralmente denominadas `a`, `b`, `c`, etc.

4.3 Valor polimórfico

Um valor é chamado **polimórfica** (*de muitas formas*) se o seu tipo contém uma ou mais *variáveis de tipo*.

Por exemplo, o tipo da função `head` pode ser escrito como

```
head :: [a] -> a
```

para qualquer tipo `a`, `head` recebe uma lista de valores do tipo `a` e retorna um valor do tipo `a`.

Já o tipo da função `length`, que recebe uma lista e resulta no tamanho da lista, é dado por:

```
length :: [a] -> Int
```

para qualquer tipo `a`, `length` recebe uma lista de valores do tipo `a` e retorna um inteiro.

A função `fst` é do tipo:

```
fst :: (a,b) -> a
```

para quaisquer tipos `a` *e* `b`, `fst` recebe um par de valores do tipo `(a,b)` e retorna um valor do tipo `a`.

4.4 Instanciação de variáveis de tipo

As variaveis de tipo podem ser *instanciadas* para diferentes tipos em diferentes circunstâncias.

Por exemplo, a função `length`

```
length :: [a] -> Int
```

pode ser aplicada em diferentes tipos listas, como mostra a tabela:

expressão	valor	instanciação da variável de tipo
`length [False,True]`	2	`a = Bool`
`length "54321"`	5	`a = Char`
`length ["ana","joel","mara"]`	3	`a = String`
`length [("ana",True)]`	1	`a = (String,Bool)`
`length [(&&),(||)]`	2	`a = Bool -> Bool -> Bool`

5 Funções polimórficas predefinidas

Muitas das funções definidas no prelúdio são polimórficas. Algumas delas são mencionadas a seguir:

```
id   :: a -> a                -- função identidade
fst  :: (a,b) -> a            -- seleciona o primeiro elemento de um par
snd  :: (a,b) -> b            -- seleciona o segundo elemento de um par
head :: [a] -> a              -- seleciona o primeiro el. de uma lista
tail :: [a] -> [a]            -- seleciona a cauda de uma lista
take :: Int -> [a] -> [a]     -- seleciona os primeiros el. de uma lista
zip  :: [a] -> [b] -> [(a,b)] -- combina duas listas, elemento a elemento
```

Observe que a lista vazia é polimórfica:

```
[] :: [a]
```

Tarefa 1

Verifique se as seguintes expressões são válidas e determine o seu tipo em caso afirmativo.

a) `['a','b','c']`

b) `('a','b','c')`

c) `[(False,'0'),(True,'1')]`

d) `([False,True],['0','1'])`

e) `[tail,init,reverse]`

f) `[[]]`

g) `[[10,20,30],[],[5,6],[24]]`

h) `(10e-2,20e-2,30e-3)`

i) `[(2,3),(4,5.6),(6,4.55)]`

j) `(["bom","dia","brasil"],sum,drop 7 "Velho mundo")`

k) `[sum,length]`

Tarefa 2

Determine o tipo de cada uma das funções definidas a seguir, e explique o que elas calculam.

a) `second xs = head (tail xs)`

b) `const x y = x`

c) `swap (x,y) = (y,x)`

d) `apply f x = f x`

e) `flip f x y = f y x`

f) `pair x y = (x,y)`

g) `palindrome xs = reverse xs == xs`

h) `twice f x = f (f x)`

i) `mostra (nome,idade) = "Nome: " ++ nome ++ ", idade: " ++ show idade`

Tarefa 3

Defina uma função chamada `ultimo` que seleciona o último elemento de uma lista não vazia, usando as funções do prelúdio.

Observação: já existe a função `last` no prelúdio com este propósito.

Tarefa 4

Defina uma função chamada `primeiros` que seleciona todos os elementos de uma lista não vazia, exceto o último., usando as funções do prelúdio.

Observação: já existe a função `init` no prelúdio com este propósito.

Tarefa 5

Usando funções da biblioteca, defina a função `metade :: [a] -> ([a],[a])` que divide uma lista em duas metades. Por exemplo:

```
> metade [1,2,3,4,5,6]
([1,2,3],[4,5,6])

> metade [1,2,3,4,5]
([1,2],[3,4,5])
```