Tuplas e Compreensão de listas

- Tuplas

- Compreensão de listas

Tuplas

- Haskell fornece outra maneira de declarar vários valores em um único tipo de dados.
- É conhecido como Tupla.
- Uma tupla pode ser considerada uma categoria de lista,
- No entanto, existem algumas diferenças técnicas entre uma tupla e uma lista.

Tuplas

- Uma Tupla é um tipo de dados **imutável**,
  - pois não podemos modificar o número de elementos em tempo de execução.

- Enquanto uma Lista é um tipo de dados **mutável**,
  - podemos alterar a quantidade de elementos durante a execução do programa.

Tuplas

- Por outro lado, Lista é um tipo de dados **homogêneo**,
  - elementos com o mesmo tipo de dados.

- mas Tupla é **heterogênea** por natureza,
  - porque uma Tupla pode conter diferentes tipos de dados dentro dela.

Tuplas

- As tuplas são representadas por parênteses simples.
- Haskell trata uma tupla da seguinte maneira:
- Prelude> (1,1,'a')
- Isso produzirá a seguinte saída:
- (1,1,'a')
- No exemplo acima, usamos uma tupla com duas variáveis do tipo numérico e uma variável do tipo char.

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.

Tuplas

- Em Haskell, uma expressão tupla é formada por uma sequência de expressões separadas por vírgula e delimitada por parênteses:
- (exp_1 ,…, exp_n)
- onde n >= 0 e n != 1, e exp_1, . . ., exp_n são expressões cujos valores são os componentes da tupla.

Tuplas

- São exemplos de tuplas:
- (‘A’, ‘H’)
- (‘G’, 4)
- (‘A’, ‘B’, ‘C’)
- (‘J’, True)
- (True, ‘A’, “malucos”)

Tuplas

- O tipo de uma tupla é o produto cartesiano dos tipos dos seus componentes.
- Sintaticamente um tipo tupla é formado por uma sequência de tipos separados por vírgula e delimitada por parênteses:
- (t_1 , ... , t_n)
- onde n >= 0 e n != 1, e t_1, . . ., t_n são os tipos dos respectivos componentes da tupla.

Tuplas

- São exemplos de tipo de tuplas:
- (Int, Int, Char)
- (Bool, Char)
- (String, Bool)
- (Char, Char, Char)

Tuplas

- Qual os tipos das seguintes tuplas ?
- (‘A’, ‘H’) —-
- (‘G’, 4) —-
- (‘A’, ‘B’, ‘C’) —
- (‘J’, True) —-
- (True, ‘A’, “malucos”) —

Tuplas

- Qual os tipos das seguintes tuplas ?
- (‘A’, ‘H’) —- (Char, Char)
- (‘G’, 4) —- (Char, Int)
- (‘A’, ‘B’, ‘C’) — (Char, Char, Char)
- (‘J’, True) —- (Char, Bool)
- (True, ‘A’, “malucos”) — (Bool, Char, String)

Tuplas

- Observe que o tamanho de uma tupla (quantidade de componentes) é codificado no seu tipo.
  - isso difere em uma lista
- ( ) é a tupla vazia, do tipo ( ) .
- **Não existe tupla de um único componente**.

Tuplas

- 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`

Tuplas

- Algumas funções 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

Tuplas

- 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)]

- Prelude> zip [1,2,3,4,5] [5,5,5,5,5]

[(1 ,5) ,(2 ,5) ,(3 ,5) ,(4 ,5) ,(5 ,5)]

- Prelude> zip [5,3,2,6,2,7,2,5,4,6,6] ["im","a","turtle"]

[(5,"im"),(3,"a"),(2,"turtle")]

Tuplas

- Definição de funções com tuplas:
- -- Função que se passa uma tupla (nome, idade) e devolve a idade
- verIdade :: (String, Int) -> Int
- verIdade (a, b) = b
- *Main> verIdade ("pedro",19)

19

Tuplas

- Vamos definir uma função que produza o quociente e o resto da divisão inteira de dois números.
- divInt :: Int -> Int -> (Int, Int)
- divInt a b = (q, r)
- where
- q = div a b
- r = rem a b
- *Main> divInt 10 5

(2, 0)

Compreensão de listas

- Em Matemática, a notação de compreensão pode ser usada para construir novos conjuntos a partir de conjuntos já conhecidos. Por exemplo,
- $\{x^2 \mid x \in [1\ldots5]\}$
- é o conjunto {1, 4, 9, 16, 25} de todos os números x^2 tal que x é um elemento do conjunto {1, 2, 3, 4, 5}.

Compreensão de listas

- Em Haskell, também há uma notação de compreensão similar que pode ser usada para construir novas listas a partir de listas conhecidas.
- Por exemplo:
- [x^2 | x <- [1..5]]
- é a lista [1,4,9,16,25] de todos os números x^2 tal que x é um elemento da lista [1,2,3,4,5].
- A frase x <- [1..5] é chamada **gerador**, já que ela informa como gerar valores para a variável x.

Compreensão de listas

- Compreensões podem ter múltiplos geradores, separados por vírgula.
- Por exemplo:
- [(x,y) | x <- [1,2,3], y <- [4,5]]
- [(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]

Compreensão de listas

- Se a ordem dos geradores for trocada, a ordem dos elementos na lista resultante também é trocada.
- Por exemplo:
- [(x,y) | y <- [4,5], x <- [1,2,3]]
- [(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]

Compreensão de listas

- Geradores múltiplos são semelhantes a loops aninhados:
- Os últimos geradores são como loops mais profundamente aninhados cujas variáveis mudam mais freqüentemente.
- No exemplo anterior, como x <- [1,2,3] é o último gerador, o valor do componente x de cada par muda mais frequentemente.

Compreensão de listas

- Geradores posteriores podem depender de variáveis introduzidas em geradores anteriores.
- Por exemplo:
- [(x,y) | x <- [1..3], y <- [x..3]]
- [(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
- é a lista [(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)] de todos os pares de números (x,y) tal que x e y são elementos da lista [1..3] e y >= x.

Compreensão de listas

- Compreensões de lista podem usar guardas para restringir os valores produzidos por geradores anteriores.
- Por exemplo:
- [x | x <- [1..10], even x]

[2,4,6,8,10]

- é a lista de todos os números x tal que x é um elemento da lista [1..10] e x é par.

Compreensão de listas

- Como exemplo, usando uma guarda podemos definir uma função para calcular a lista de divisores de um número inteiro positivo:
- divisores :: Int -> [Int]
- divisores n = [x | x <- [1..n], mod n x == 0]
- Exemplos de aplicação da função:
- divisores 15

[1,3,5,15]

- divisores 120

[1,2,3,4,5,6,8,10,12,15,20,24,30,40,60,120]

Compreensão de listas

- Um número inteiro positivo é primo se seus únicos divisores são 1 e ele próprio.
- Assim, usando divisores, podemos definir uma função que decide se um número é primo:
- primo :: Int -> Bool
- primo n = divisores n == [1,n]
- Exemplos de aplicação da função:
- primo 15

False

- primo 7

True

Compreensão de listas

- Usando um guarda agora podemos definir uma função que retorna a lista de todos os números primos até um determinado limite:
- primos :: Int -> [Int]
- primos n = [x | x <- [2..n], primo x]
- Exemplos de aplicação da função:
- primos 40

[2,3,5,7,11,13,17,19,23,29,31,37]

- primos 12

[2,3,5,7,11]

APS 7

- Crie o programa fonte aula09.hs (com todas as funções e variáveis) como codificado nesta aula. Teste todas as funções do programa fonte, carregando no GHCi.
- Observe a necessidade de definir os tipos de dados das variáveis e também das funções.
- Essa APS não precisa ser enviada para o professor, mas deve ser realizada.