haskell - livro

Tuplas, no entanto, são usadas quando você sabe exatamente quantos valores você quer combinar e o seu tipo depende de quantos componentes ela tem e os tipos dos componentes. Tuplas são caracterizadas por parênteses com seus componentes separados por vírgulas.Outra diferença fundamental é que elas não precisam ser homogêneas. Ao contrário de uma lista, uma tupla pode conter uma combinação de vários tipos.

fst recebe um par e retorna seu primeiro componente.

```
ghci> fst (8,11)
8
ghci> fst ("Wow", False)
"Wow"
```

snd recebe um par e retorna seu segundo componente. Surpresa!

```
ghci> snd (8,11)
11
ghci> snd ("Wow", False)
False
```

função somente utilizada em PARES

Ao contrário de listas, cada valor da tupla tem seu tipo. Então a expressão (True, 'a') tem o tipo (Bool, Char), e uma expressão como ('a','b','c') deverá retornar (Char, Char, Char). 4 == 5 sempre retornará False, que é do tipo Bool.

CHAR sinonimo de string!

Float é um número real em ponto flutuante de precisão simples.

Double é um número real em ponto flutuante com o dobro(!) de precisão.

Bool é um tipo booleano. Pode ter apenas dois valores: True ou False.

Char representa um caractere. É delimitado por aspas simples. Uma lista de caracteres é denominada String.

```
⇒
```

"exige que..." (restrição de tipo) em declarações de tipo

```
ghci> :t (==)
(==) :: (Eq a) ⇒ a → a → Bool
```

Nota: o operador de igualdade (==) é uma função. Assim como +, *, -, / e quase todos os outros operadores.

```
factorial :: (Integral a) ⇒ a → a
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

```
bmiTell :: (RealFloat a) ⇒ a → a → String
bmiTell weight height
    | bmi <= skinny = "Você esta abaixo do peso!"
    | bmi <= normal = "Supostamente você esta normal. Pfff, aposto que você
é feio!"
    | bmi <= fat    = "Você esta gordo! Faça uma dieta, gorducho!"
    | otherwise     = "Você é uma baleia, meus parabéns!"
    where bmi = weight / height ^ 2
          skinny = 18.5
          normal = 25.0
          fat = 30.0
```

A palavra-chave `where` em Haskell serve para **definir variáveis auxiliares** (ou até funções) que **só existem dentro da função principal**. fica no **final** da função

Use `let` quando estiver **dentro de expressões**, como `if`, listas, ou até no `do` (em monads).

Associações let dão valores a funções e também são expressões, mas tem um escopo mais restrito por não serem acessíveis dentro de guards.

```
cylinder :: (RealFloat a) ⇒ a → a → a
cylinder r h =
    let sideArea = 2 * pi * r * h
        topArea = pi * r ^2
    in  sideArea + 2 * topArea
```

CASE: teste de possibilidades

```
case valor of
padrão1 → resultado1
padrão2 → resultado2
_       → resultadoPadrão
```

```
descreveLista :: [Int] → String
descreveLista xs = case xs of
[]      → "Lista vazia"
[x]     → "Tem um elemento"
[x, y]  → "Tem dois elementos"
_       → "Tem mais de dois elementos"
```

RECURSÃO

```
maximum' :: (Ord a) ⇒ [a] → a
maximum' [] = error "maximum of empty list"
maximum' [x] = x
maximum' (x:xs)
    | x > maxTail = x
    | otherwise = maxTail
    where maxTail = maximum' xs
```

```
take' :: (Num i, Ord i) ⇒ i → [a] → [a]
take' n _
```

```
    | n <= 0   = []
take' _ []     = []
take' n (x:xs) = x : take' (n-1) xs
```

```
reverse' :: [a] -> [a]
reverse' [] = []
reverse' (x:xs) = reverse' xs ++ [x]
```

```
zip' :: [a] -> [b] -> [(a,b)]
zip' _ [] = []
zip' [] _ = []
zip' (x:xs) (y:ys) = (x,y):zip' xs ys
```

```
elem' :: (Eq a) => a -> [a] -> Bool
elem' a [] = False
elem' a (x:xs)
    | a == x    = True
    | otherwise = a `elem'` xs
```

Bastante simples e previsível. Se a cabeça não é o elemento, então nós checamos a cauda. Se alcançarmos uma lista vazia, então o resultado é False.

O `quicksort` é um **algoritmo de ordenação** — ele pega uma lista desorganizada e **coloca os elementos em ordem crescente**

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
    let smallerSorted = quicksort [a | a <- xs, a <= x]
        biggerSorted = quicksort [a | a <- xs, a > x]
    in  smallerSorted ++ [x] ++ biggerSorted
```

Em Haskell, **recursão é o principal jeito de repetir coisas**. E ela sempre segue esse **padrão mágico**:

✨ Padrão da Recursão:

1. **Caso base (ou caso limite)**

   → quando a função **para**, sem chamar a si mesma de novo.

2. **Caso recursivo**

   → quando a função **faz algo com o primeiro elemento** e depois **se aplica ao resto**.