Tipos básicos do haskell

Tipo	Descrição
Int	Inteiros
Float	Ponto flutuante de precisão simples
Double	Ponto Flutuante de precisão dupla
Char	Caractere
String	Lista de caracteres.
Bool	Um valor booleano que tem que ser necessariamente “True” ou “False”, ou seja, não aceita atribuição de “0” ou “1”.

NOTE : O valor que deve ser atribuído ao tipo “Bool” tem que ser necessariamente “**T**rue” ou “**F**alse”, ou seja, a primeira letra do valor verdade tem que ser necessariamente maiúscula. Caso algum aventureiro tente atribuir “true” ou “false” (com as primeiras letras minúsculas), ganhará um erro.

Operadores aritméticos :

Operador	Descrição
+	adição
-	subtração
*	multiplicação
/	Divisão com retorno de ponto flutuante
^	Potenciação
**	Potenciação

A diferença entre a primeira e a segunda forma de performar uma operação de potenciação é sutil. O operador “**” funciona tanto com operandos inteiros, quanto com decimais e sempre retorna um decimal, já o operador “^” só trabalha com expoentes inteiros e retorna um inteiro ou um decimal a depender do tipo da base.

Além destes operadores, ainda existem 2 funções aritméticas muito úteis, são elas :

- mod : usado para obtenção do resto da divisão de dois números.

```
5 `mod` 2
        1
```

- div : usado para obtenção do resultado inteiro da divisão

```
7 `div` 2
        3
```

NOTE : Caso você esteja lidando com números negativos em operações aritméticas de multiplicação, divisão, exponenciação e etc, assegure-se que esses operandos estejam entre parênteses. Caso contrário, ocorrerá um erro. Por exemplo, se testássemos o seguinte código no GHCI :

```
2 * -5
```

Isto geraria um “precedence parsing error”. No entanto, se o “-5” estivesse envolto por parênteses, nenhum erro ocorreria.

NOTE: Apesar de o haskell ser fortemente tipado, operações entre números inteiros e de ponto flutuante são tolerados. O exemplo abaixo ilustra bem isto.

```
12.0 + 7
```

Operadores lógicos

Operador	Descrição
&&	conjunção
||	disjunção
not	negação

Operadores relacionais

Operador	Descrição
==	Igualdade
/=	Diferente
<	Menor que
<=	Menor ou igual que
>	Maior que
>=	Maior ou igual que

Definição de funções

Cada função em haskell é composta por uma assinatura e um escopo. A seguir, define-se a sintaxe da assinatura.

<nomeFuncao> <::> {<tipo> {->}}

Logo após o símbolo "::" os tipos de dados que a função processará serão passados e eles serão separados pelo símbolo "->". O processo de declaração dos tipos culmina na declaração do tipo de retorno da função, ou seja, após toda a "lista dos tipos dos argumentos" ter sido passada, ainda adiciona-se um "->" a mais acompanhando o tipo retorno da função. Caso a função não receba parâmetros, basta pôr o tipo de retorno da função desacompanhada da ceta.

Observe o exemplo da declaração da função "somaUm" para entender melhor.

```
1. somaUm :: Int -> Int
```

Essa função recebe um decimal qualquer (o primeiro "Float") e retorna como output um outro decimal (o segundo "Float").

Implementação de funções

Após feita a declaração da função, basta implementá-la. Observe :

```
1. somaUm :: Int -> Int
2. somaUm minhaVariavel = minhaVariavel + 1
```

No exemplo acima "minhaVariavel" é a entrada da função e "minhaVariavel + 1" é o valor de retorno. Observe um exemplo mais complexo abaixo :

```
1. calculaDeltaEquacaoQuadratica :: Int -> Int -> Int -> Int
2. calculaDeltaEquacaoQuadratica a b c = b^2 - 4 * a * c
```

Para chamar estas funções no GHCI é muito simples, observe :

```
1. >> somaUm 2
        3
2. > > calculaDeltaEquacaoQuadratica 1 (-4) (-5)
        36
```

Além destes exemplos simples, em virtude do seu caráter funcional, o haskell nos permite fazer chamadas de funções mais interessantes que essas, como as listadas a seguir:

```
1. >>calculaDeltaEquacaoQuadratica (somaUm 22) (somaUm 12) (somaUm 56) --1°exemplo
        -5075
2. >>somaUm (calculaDeltaEquacaoQuadratica 8 10 3) --2° exemplo
        5
3. >>(calculaDeltaEquacaoQuadratica (somaUm 22) (somaUm 12) (somaUm 56)) + (somaUm
(calculaDeltaEquacaoQuadratica 8 10 3)) --3° exemplo
        -5070
```

No primeiro exemplo temos :

a = somaUm 22
b = somaUm 12
c = somaUm 56

De forma análoga, o único parâmetro passado durante a chamada de “somaUm” na 2° linha foi “calculaDeltaEquacaoQuadratica 8 10 3”. E a 3° linha, foi a mais interessante, pois “+” recebe dois operandos, onde cada um corresponde a uma função. Observe-os abaixo :

1. (calculaDeltaEquacaoQuadratica (somaUm 22) (somaUm 12) (somaUm 56))
2. (somaUm (calculaDeltaEquacaoQuadratica 8 10 3))

Além disto, ainda é possível definir novas funções em virtude de funções já existentes.

```
1. multiplicaDoisSomaDois :: Float -> Float
2. multiplicaDoisSomaDois minhaVariavel = somaUm minhaVariavel + somaUm minhaVariavel
```

```
1. multiplicaDoisSomaDois 5
        >> 12
```

NOTE: Apesar de todas as funções terem sido implementadas corretamente, perceba que elas necessariamente precisam receber dados do tipo “Int” para funcionar, seria mais desejoso fazer com que elas recebessem qualquer tipo de parâmetro numérico através da omissão das suas assinaturas, pois, uma vez que o haskell possui um sistema de inferência de tipos bastante sofisticado, ele é capaz de deduzir os tipos dos parâmetros através da forma com que eles são usados nas funções. Nos exemplos posteriores que trabalhem com essas funções suponha que as suas assinaturas foram omitidas e para mais detalhes sobre o sistema de inferência de tipos do haskell, leia o apêndice “A”.

NOTE: Todos os parâmetros com sinal de menos precisam ser postas entre parênteses, caso contrário, um erro ocorrerá. Por esta razão, no 3° bloco de código alguns dos parâmetros de “calculaDeltaEquacao Quadratica”estão entre parênteses.

NOTE: Perceba que, diferente da maioria das linguagens – que separam os parâmetros por vírgulas – o haskell usa espaços para realizar essa separação.

NOTE: O haskell não permite que o nome das funções comecem com letras maiúsculas.

Funções prefixas e infixas

O haskell dispõe de dois tipos de funções, as prefixas e as infixas. Ambas desempenham basicamente o mesmo papel : receber zero ou mais parâmetros e retornar valores. A única coisa que as diferencia é forma como elas recebem os parâmetros. Enquanto os parâmetros são passados após o nome da função nas prefixas, nas infixas eles são passados antes **e** depois do nome da função. Isto implica dizer que somente é possível usar funções infixas caso elas tenham apenas 2 parâmetros, pois seria incrivelmente complicado de trabalhar com o elas caso contrário ! Observe mais detalhadamente a sintaxe delas.

- **função prefixa :** <nome da função> <parametros>
  - Exemplos são listados abaixo :
    - somaTodosOsCinco 7 10 12 14 9
      - 52
    - resolveBhaskaraX1 2 (-6) (-8)
      - -1
    - encontraHipotenusa 21 28
      - 35
- **Função infixa :** <parametro> <nome da função> <parametro>
  - Exemplos :
    - 7 + 7
      - 14
    - "oi" == "hello"
      - False
    - 7 > 6
      - True

Caso você não tenha percebido ainda, a definição de função infixa que foi dada, nos induz a acreditar que os operadores aritméticos, lógicos e relacionais são, na verdade, funções ! De fato, esta suspeita não está sem fundamentos, pois, à medida que cada um dos operadores que foram apresentados só recebem 2 parâmetros e retornam um valor, esta dedução se mostra correta. Todos os operadores em haskell são, *na verdade*, funções. Por via de regra, tudo em haskell é funcional.

Além disto, ainda é possível adaptar funções prefixas para que elas funcionem como infixas, caso elas tenham apenas dois parâmetros. Considere o exemplo abaixo.

```
1. minhaFuncao parametro1 parametro2 -- forma prefixa
2. parametro1 `minhaFuncao` parametro2 -- forma infixa
```

Ou seja, uma função prefixa que é posta entre crases atua como uma infixa, desde que seus parâmetros sigam a sintaxe descrita. Esta adaptação, quando feita de forma oportuna e pertinente, pode melhorar significativamente a legibilidade dos códigos. Por exemplo, nos códigos em que as funções “mod” e “div” foram apresentadas, elas foram escritas de forma infixa, ou seja, as suas formas originais são:

```
1. mod 5 2
        1
2. div 7 2
        3
```

De forma semelhante, é possível adaptar uma função infixa para ser sintaticamente similar a uma prefixa. Observe o exemplo :

```
1.parametro1 minhaFunção2 parametro2 --forma infixa
2.(minhaFunção2) parametro1 parametro2  --forma prefixa
```

Observa-se claramente que ao pôr “minhaFuncao2” entre parênteses é possível utilizá-la de forma prefixa. Abaixo, são listados mais alguns exemplos.

```
1. (+) 7 7 --forma prefixa
        14
2.  (>) 14 2
        True
3. (&&) ((>) 12 6) ((>) 40 13) -- corresponde a : 7 > 2 && 10 > 1
        True
```

Algumas funções úteis

- min : Recebe dois números, sejam eles inteiros ou de ponto flutuante, e retorna o menor entre os dois.

```
1. min 4 10
        4
```

- max : Tem uma proposta semelhante à da função “min”, mas retorna o maior elemento.

```
1. max 10 4
        10
```

If-then-else

A usabilidade da expressão if-then-else em haskell se assemelha muito a do operador ternário em C, isto pois ele não é usado com frequência para delimitar fluxos de execução complexos, como é comum de se esperar de uma expressão “if-them-else”, mas é tipicamente usado para a atribuição de valores a variáveis através de condições simples, exatamente como um operação ternária. Observe as implementações de min’ e max’.

```
1. min' a b = if a < b then a else b
2. max' a b = if a > b then a else b
```

Ainda é possível deixar o código mais elegante através do uso de indentação.

```
1.min' a b =
2.  if a < b
3.     then a
4.  else
5.     b
```

É necessário tomar um cuidado especial ao definir uma função que tenha uma expressão if-then-else. Observe :

```
1. minSomaUmA a b = if a < b -- esta forma de declaração também está correta
2.                           then a + 1
3.                           else b
4. minSomaUmAB a b = (if a < b
5.                            then a
6.                            else b)+1
```

Na primeira função, somente se o “a” for menor soma-se 1 ao resultado, mas na segunda função sempre será adicionado 1 ao resultado e isto acontece porque os parênteses estão cercando toda a expressão if-then-else.

Além disto, ainda é possível usar um “else if”, observe :

```
1.max" a b c =
2.  if a > b && a > c
3.    then a
4.  else if b > a && b > c
5.    then b
6.  else
7.    c
```

NOTE: Em haskell o símbolo do apóstrofo é um caractere válido para nomes de funções. Frequentemente é usado para expressar uma reimplementação.

NOTE: Caso você opte por escolher o espaço para a indentação, o haskell lhe permite escolher quantos usar, desde que a utilização seja uniforme durante todo o código. O mais comum são 2 ou 4 espaços. Além de que caso seja escolhido o espaço, não poderá ser usado a tabulação e vice-versa durante todo o código.