Capítulo 2
Listas em haskell

Um dos tipos de dados mais importantes de qualquer linguagem funcional são as listas. Portanto, é indispensável compreender bem este tipo de dado.

Detalhes importantes

- Todas as listas em haskell são homogêneas, ou seja, armazenam apenas um tipo de dado. Portanto, caso algum aventureiro tente inserir um caractere em uma lista de inteiros, isto gerará um erro.
- Strings são, em essência, listas de caracteres

Mais ou menos um “Olá mundo”

Para criar uma função que retorne uma lista de caracteres é muito simples, observe :

```
1. maisMenosOlaMundo :: [Char]
2. maisMenosOlaMundo = ['O', 'l', 'a', ' ', 'm', 'u', 'n', 'd', 'o']
```

Caso você tenha medo de listas de caracteres por algum trauma em “fundamentos da programação”, você até pode declarar o tipo de retorno da função como “String”, basta não se esquecer que elas são tipos equivalentes.

```
1. maisMenosOlaMundo :: String
2. maisMenosOlaMundo = "Ola mundo"--esta forma de implementação também daria certo com [Char]
```

Caso você invoque essa função no ghci, você irá “imprimir”[1] seu primeiro “Olá mundo” em haskell !

NOTE : Uma vez que strings são listas de caracteres, todas as funções que se aplicam as listas também se aplicam as strings.

Funções triviais

1. **Concatenação(++)** : Concatenar listas significa combinar uma ou mais listas em uma única lista. Observe o exemplo à baixo :

```
1. listaConcatenada :: String
2. listaConcatenada = “Olá” ++ “ ” ++ “mundo”
```

1 Não se trata propriamente de uma impressão, pois não há a chamada de uma função de output.

2. **Pré-anexação(:)** : O ato de adicionar um elemento ao início da lista. No caso das strings, este operador recebe um Char e o anexa ao início de um [Char], portanto tentar usá-lo para anexar um [Char] a um outro [Char] será em vão.

```
listaPreAnexada :: [Char]
listaPreAnexada = 'O':'l':'a':" mundo"
```

Apesar de a concatenação e a pré-anexação serem bastante similares, a concatenação é mais lenta, pois ela funciona percorrendo toda a lista para depois adicionar os novos elementos a ela, enquanto a pré-anexação é “instântanea”.

Observe o código a seguir para notar algo interessante.

```
1. listaPreAnexadaContinuamente :: String
2. listaPreAnexadaContinuamente = ‘A’:‘B’:‘C’:‘D’: ‘E’:[]
```

Note que para “declarar” uma lista vazia, basta pôr um par de colchetes vazios !

3. **Comparação** (>, <, >=, <=, /=, ==) : A comparação entre listas (maior e menor) segue uma lógica lexicográfica, tanto para strings, quanto para listas de tipos numéricos, ou seja, o 1° elemento da primeira lista será comparado com o 1° elemento da segunda lista, de tal forma que será avaliado qual dos dois é o maior/menor. Assumindo que os dois sejam diferentes, o resultado oriundo da comparação será retornado. Caso os dois sejam iguais, haverá a comparação entre o 2° elemento da primeira lista e o 2° elemento da 2° lista, assim suscetivamente.

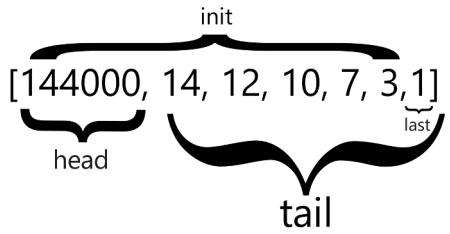
```
1. palavra1 = "Felicidade"
2. palavra2 = "Perpetua"
> > palavra1 > palavra2
        False
```

```
1. lista1 = [7,10,14,3]
2. lista2 = [6,100,100,100]
> > lista1 > lista2
        True
```

```
1. lista1 = [144000, 7, 10, 14, 3]
2. lista2 = [144000, 6, 100, 100,100]
> > lista1 > lista2
        True
```

Funções elementares

1. Head, tail, last, init : A melhor forma de explicar essas funções é através de uma imagem simples :



Explicando brevemente, a função “head” retorna o primeiro elemento da lista, “last” o último, “tail” a lista sem o 1° elemento, e “init” a lista sem o último elemento. Observe o uso dessas funções abaixo.

```
1. goodNumbers = [144000, 14, 7, 10, 14, 3]
2. >>head goodNumbers
        144000
3.>>last goodNumbers
        3
4. init goodNumbers
        [144000,14,7,10,14]
5. tail goodNumbers
        [14,7,10,14,3]
```

2. length : retorna o tamanho de uma lista.

```
1. goodNumbers = [144000, 14, 7, 10, 14, 3]
2. >>length goodNumbers
        6
```

3. null : retorna um valor booleano indicando se uma lista está ou não vazia.

```
1. goodNumbers = [144000, 14, 7, 10, 14, 3]
2. myEmptyList = []
3. >> null goodNumbers
        False
4. >> null myEmptyList
        True
```

4. reverse : Retorna uma lista de forma invertida.

```
1. goodNumbers = [144000, 14, 7, 10, 14, 3]
2. >>reverse goodNumbers
        [3,14,10,7,14,144000]
```

5. take : recebe uma lista e um inteiro “n” e retorna os “n” primeiros elementos dessa lista.

```
1. goodNumbers = [144000, 14, 7, 10, 14, 3]
2. >>take 3 goodNumbers
        [144000,14,7]
```

Perceba que, caso o valor de “n” seja maior que o tamanho da lista, toda a lista será retornada.

```
1. goodNumbers = [144000, 14, 7, 10, 14, 3]
2. >>take 12 goodNumbers
        [144000,14,7,10,14,3]
```

6. drop : recebe uma lista e um inteiro “n” e retorna os “n” últimos elementos dessa lista.

```
1. goodNumbers = [144000, 14, 7, 10, 14, 3]
2. >>drop 3 goodNumbers
        [10,14,3]
```

Perceba que, caso o valor de “n” seja maior que o tamanho da lista, uma lista vazia será retornada.

7. maximum : retorna o maior elemento da lista.

```
1. goodNumbers = [144000, 14, 7, 10, 14, 3]
2. >>maximum goodNumbers
        144000
```

8. minimum : retorna o menor elemento da lista

```
1. goodNumbers = [144000, 14, 7, 10, 14, 3]
2. >>minimum goodNumbers
        3
```

9. sum : retorna a soma de todos os elementos da lista.

```
1. goodNumbers = [144000, 14, 7, 10, 14, 3]
2. >>sum goodNumbers
        144048
```

Do contrário do que se é usual, caso a lista seja uma string, a função “sum” não retornará o somatório dos códigos ASCII delas, mas sim um erro, pois o haskell usa codificação “unicode”, e não ASCII.

10. product : retorna o produto de todos os elementos da lista.

```
1. goodNumbers = [144000, 14, 7, 10, 14, 3]
2. >>product goodNumbers
        5927040000
```

11. elem : recebe uma lista e uma valor “x” e retorna um booleano indicando se “x” está na lista

```
1. goodNumbers = [144000, 14, 7, 10, 14, 3]
2. >>elem 6 goodNumbers
        False
```

Ranges

Em haskell, range é um recurso usado para gerar listas cujos elementos estejam dentro de um intervalo fechado estabelecido.

```
1. up2Seven = [1..7]
2. >>up2Seven
        [1,2,3,4,5,6,7]
```

É possível fazer algo semelhante com strings também :

```
1. up2H = [‘A’ .. ‘H’]
2. >>up2H
        "ABCDEFGH"
```

Como é de se esperar, quando se utiliza ranges para gerar uma string, o haskell se baseia na codificação unicode para tanto e, uma vez que a codificação ASCII está contida na unicode, logo se houvesse um range partindo do “A” (maiúsculo) ao “z” (minúsculo), uma vez que letras maiúsculas são “menores” que letras minúsculas na codificação ASCII, o resultado seria uma lista com alguns caracteres não alfabéticos.

```
1. a2z = [‘A’ .. ‘z’]
2. >>a2z
"ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`
abcdefghijklmnopqrstuvwxyz"
```

Além disto, ainda é possível especificar de quantos em quantos elementos deseja-se “pular” ao usar os ranges através de uma pequena variação da sintaxe já conhecida, observe :

[<começo>, <próximo elemento> .. <destino>]

Desta forma, a quantidade de elementos que serão “pulados” a partir de “começo” corresponde a “próximo elemento” – “começo” - 1. Observe o exemplo abaixo.

```
1. allOddsUp2forty = [1, 3 ..40]  --imprime todos os elementos impares de 1 até 40
2. >> allOddsUp2forty
        [1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39]
```

É necessário ter um grande cuidado ao trabalhar com “ranges” de números de ponto flutuante, pois, com o tempo, ocorrerão erros de precisão. Observe :

```
1. up2two = [0.1, 0.32 .. 2]
2. >> up2two

[0.1,0.32,0.54,0.76,0.98,1.2000000000000002,1.4200000000000002,1.6400000000000001,1.86,2.08]
```

compreensão de listas

É uma técnica que permite a criação de novas listas a partir de outras e a filtragem meticulosa de elementos de listas pré existentes. Por exemplo, imagine ser necessário gerar uma lista dos quadrados de todos os números de 1 à 20[2], para tanto, o seguinte código poderia ser usado :

```
1. squaresUp2Twenty = [x^2 | x <- [1..20]]
2. >> squaresUp2Twenty
        [1,4,9,16,25,36,49,64,81,100,121,144,169,196,225,256,289,324,361,400]
```

Em um linguajar um pouco mais matemático, a notação da linha 1 poderia ser facilmente traduzida como “x ao quadrado, tal que x pertence ao conjunto dos números de 1 à 20”.

Como dito antes, além da criação de novas listas, ainda é possível realizar filtragem de elementos. Por exemplo, caso fosse necessário selecionar todos os elementos “x” de 1 à 100 de tal forma que o quadrado de “x” não exceda a razão do somatório de todos os elementos de 1 à 100 por “x”, o código abaixo solucionaria o problema.

```
1. meticulousFilter = [x | x <- [1..100], x**2 < (sum [1..100])/x]
2. >>  meticulousFilter
        [1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0,11.0,12.0,13.0,14.0,15.0,16.0,17.0]
```

Perceba que na linha 1, além de especificar o conjunto ao qual o “x” pertence, ela também fornece um predicado. Mais uma vez, ao traduzir esta função em uma linguagem mais matemática, teríamos “‘x’, tal que ‘x’ pertence ao conjunto de números de 1 à 100 cujo quadrado não excede a razão do somatório de 1 à 100 por ‘x’ ”

2 Isto está bastante próximo da conceitualização de um laço “for”.

Assim como os “ranges” não se limitam somente a tipos numéricos, as compreensões de listas também não se limitam, uma vez que são capazes de funcionar com strings também, observe :

```
1. somenteLetras = [ myStr | myStr <- ['A'..'z'], myStr < '[' || myStr > '`']
2. >>  somenteLetras
        "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"
```

Se relembrarmos a função “a2z” – usada para exemplificação de ranges com strings – notamos que entre as letras maiúsculas e minúsculas há alguns caracteres não alfanuméricos, como “\” e “^”, sendo o primeiro “[” e o último “`”. Tendo essas informações, é possível filtrar todos os elementos alfanuméricos do intervalo de “A” à “z” usando compreensão de listas, basta expressar que somente são desejadas letras maiúsculas (caracteres com o código ASCII menor que o de “[”) e minúsculas (caracteres com o código ASCII maior que o de “`”).

Tuplas

Tuplas são, tipicamente, um par de dados, por exemplo, (1,3). Perceba que não necessariamente o 1° e o 2° elemento precisam ser do mesmo tipo, logo os seguintes exemplos são sintaticamente corretos : (7, True), (6, False). Abaixo são listadas algumas funções úteis que envolvem tuplas :

1. fst : retorna o primeiro elemento da tupla; é um mnemónico para “first” – do inglês “primeiro”

```
1. myTuple = (10,14)
2. > > fst myTuple
        10
```

2. snd : retorna o segundo elemento da tupla; é um mnemónico para “second” – do inglês segundo

```
1. myTuple = (7,"perfection")
2. > > snd myTuple
        "perfection"
```

3. zip : É usada para combinar 2 listas em uma única lista de tuplas. Caso as listas fornecidas tenham tamanhos diferentes, a quantidade de tuplas na lista gerada corresponderá ao tamanho da menor lista fornecida.

```
1. letrasMinusculasMaiusculas = zip ['a'..'z'] ['A' .. 'Z']
2. > > letrasMinusculasMaiusculas
[('a','A'),('b','B'),('c','C'),('d','D'),('e','E'),('f','F'),('g','G'),('h','H'),('i','I'),('j','J'),('k','K'),('l','L'),
('m','M'),('n','N'),('o','O'),('p','P'),('q','Q'),('r','R'),('s','S'),('t','T'),('u','U'),('v','V'),('w','W'),
('x','X'),('y','Y'),('z','Z')]
```

```
1. tuplaNumerica = zip [1..5] [6 .. 1000]
2. > > tuplaNumerica
        [(1,6),(2,7),(3,8),(4,9),(5,10)]
```

Uma outra coisa a ser notada a respeito das tuplas é que elas não se limitam a ter apenas 2 elementos, mas elas podem ter “n” elementos ! Observe :

```
1. myTriple = (1,3,7)
```

Apesar de tuplas de mais de dois elementos terem a sua utilidade para a resolução de problemas mais complexos, as tuplas de dois elementos usadas com maior frequência.