Introdução ao Haskell

Nota histórica

Durante uma conferência em 87 foi decidido que era necessário um standard de linguagem funcional. O Haskell nasceu aí e foi evoluindo aos poucos até em 97 ter sido elaborada a definições base da linguagem, este marco ficou conhecido como Haskell 98.
Sendo uma linguagem funcional, destaca-se imediatamente pela inexistência de ciclos e em vez disso é dada preferência ao uso da recursividade.
Entre outras características, que podem ser facilmente pesquisadas, Haskell, pode-se considerar *Strongly Typed*, *Pure*, *Lazy evaluation*.
Para a utilização dos programas de código fonte Haskell, existem duas opções. O uso de interpretadores como por exemplo **GHC**, **Hugs**, ou através da compilação do código, sendo que neste caso utiliza-se o compilador **GHC**.

O básico

A extensão dos ficheiros Haskell é. *hs*, e cada ficheiro de código Haskell deve conter uma "etiqueta" na primeira linha que o identifica

```
module Nome where
```

Sendo que Nome, é o nome que se deseja dar, onde:

1. Se inicia obrigatoriamente por letra maiúscula;
2. Não podem existir espaços;

Em Haskell os comentários poderão ser feitos usando

```
--
```

Primeiro Programa

O nosso primeiro programa será um incrementador de um valor. Devido à necessidade de ter conhecimentos mais avançados para se poder fazer a impressão de texto, não poderemos começar pelo típico *Hello World*. Iremos também usar sempre o interpretador **GHC**.

Definição de uma função:

```
incrementa :: Int -> Int
```

Esta linha representa a assinatura de uma função, e a sua notação é inspirada na notação matemática f: X → Y. Deste modo, e fazendo correspondência:

- *incrementa* - nome da função, equivalente ao f;
- *::* - separador, equivalente a *:* numa função matemática;

- *Int* - é o domínio (X) e o contradomínio (Y) de uma função, sendo assim refere-se ao tipo de dados recebidos/resultantes;
- - > - separador de variáveis;

O programa em si:

```
incrementa :: Int -> Int
incrementa x = x+1
```

Na função *incrementa*, apresentada acima, a primeira linha de código refere-se à definição do tipo da função, isto é, refere de que tipo será o input e o output. A segunda, por sua vez, já define quais são as variáveis a ser usadas e a operação a realizar.

Um outro exemplo simples, pode ser a definição de uma função que receba dois elementos, e como resultado dê o valor da sua soma.

```
1:soma :: Int -> Int -> Int
2:soma a b = a+b
```

Listas

Um dos tipos de dados mais populares das linguagens de programação funcional, são as vulgarmente chamadas *listas* ou *sequências*. Sendo assim torna-se importante saber o que é na realidade uma lista.
Uma lista não é mais que um conjunto de "qualquer coisa", sendo que estes qualquer coisa podem ser Inteiros, Bools, Caracteres, pares, etc.
Desta forma para se poder definir uma lista, usa-se:

- "[" - usado para iniciar uma lista;
- "]" - usado para terminar uma lista;
- "," - usado para separar os vários elementos de uma lista;

Exemplo:

```
[1,2,3,4,5,6,7,8,9,10]
```

Esta é a definição da lista que contém todos os números de 1 a 10.

O exemplo apresentado é bastante simples e está correcto. No entanto torna-se pouco funcional para listas de muitos elementos.
Imagine-se então o caso em que se teria a necessidade de definir uma lista que contivesse todos os números de 1 a 100. Para tal, o melhor método seria utilizar a função *EnumFromTo*, que dentro de uma lista se pode representar por "..", e que como resultado nos dá a lista de todos os elementos de 1 a 100.

```
[1..100]
```

Os exemplos apresentados permitem a criação de listas caso os seus elementos sejam seguidos, mas por vezes as listas não contêm elementos seguidos, mas sim por exemplo só os números pares, ou só alguns caracteres. Para isto é possível utilizar *listas por*

compreensão, que são um tipo específico de listas, onde existe uma ou mais variáveis e onde podem existir "condições" de filtragem.

Exemplo_1:

```
[x|x<-[1..10], 0==mod x 2]
```

O exemplo_1, apresenta a lista de todos os elementos pares de 1 a 10.
Como se pode verificar pelo exemplo, esta lista pode como que ser dividida em duas partes: A primeira

- [x|x←[1..] - onde é apresentada a variável, e os valores a percorrer;

A segunda

- 0==mod x 2 - onde é colocada a condição, o filtro que define qual será o output. Neste caso só os valores pares eram aceites.

Por sua vez, é também possível definir uma lista de pares que contenham alternadamente Inteiros e Caracteres:

Exemplo_2:

```
[(x,y)|x<-[1..6], y<-['E'..'J']]
```

Nota: o output destas duas definições será respectivamente

```
[2,4,6,8,10]
[(1,'E'),(1,'F'),(1,'G'),(1,'H'),(1,'I'),(1,'J'),(2,'E'),(2,'F'),(2,'G
'),(2,'H'),(2,'I'),(2,'J'),(3,'E'),(3,'F'),(3,'G'),(3,'H'),(3,'I'),(3,
'J'),(4,'E'),(4,'F'),(4,'G'),(4,'H'),(4,'I'),(4,'J'),(5,'E'),(5,'F'),(
5,'G'),(5,'H'),(5,'I'),(5,'J'),(6,'E'),(6,'F'),(6,'G'),(6,'H'),(6,'I')
,(6,'J')]
```

Recursividade

Durante os anos 30, Alan Turing, definiu a ideia de máquina de Turing, que viria a ficar associada à noção de função computável (não explicarei o que é uma função computável pois teria que entrar no âmbito da lógica e da matemática). Mais tarde, o próprio Turing provou ser necessária recursividade a linguagem funcional, para nela se poder definir funções computáveis. Deste modo nasceu a recursividade, que não é mais que a invocação de uma função dentro dela própria.

```
incrementa :: Int -> Int
incrementa x = incrementa x+1
```

O caso apresentado representa uma função que incrementa uma unidade à variável x, infinitamente. No entanto esta função em si não nos é nada útil, pois nunca acaba. É possível então, colocar condições que a façam terminar.

```
incrementa :: Int -> Int -> Int
incrementa a b = if(a<b)
                 then incrementa (a+1) b
                 else a
```

Neste exemplo a função incrementa invoca-se a ela própria enquanto o valor de *a* for menor que *b*.

Recursividade com Listas

```
1:ultimo :: [Int] -> Int
2:ultimo [] = 0
4:ultimo (h:[]) = h   ou apenas ultimo [h] = h
5:ultimo (h:t) = ultimo t
```

A função apresentada percorre a lista uma vez, e como resultado dá o último elemento da lista. Para isso ser possível foi necessária a definição de três "condições":

1. ultimo [] = 0 - se a lista recebida for vazia, a função dá como resultado o valor 0;
2. ultimo (h:[]) = h - caso a lista só tenha um elemento, ela dá esse valor como resultado;
3. ultimo (h:t) = ultimo t - a função invoca-se a ela própria, passando como parâmetro sempre a cauda da lista. Vai fazendo isto até que a lista só tenha um elemento, aí salta para o caso anterior.

Nota: A variável h, refere-se à cabeça da lista (head), t refere-se à cauda (tail) da mesma lista. Para a manipulação de listas é então sempre necessário o uso de uma cabeça e de uma cauda, onde o primeiro elemento se refere sempre à cabeça, e os restantes (quer existam ou não) se referem à cauda. Sendo então necessário um operador que faça a separação entre esses elementos. Em Haskell o operador encarregue de tal função é " *:* ". Caso uma lista não contenha qualquer elemento, ela é representada por *[]*.

Acumuladores

Os acumuladores em Haskell, são um meio de se obter ciclos. Estes são valores passados durante a execução recursiva da função.

Eles têm um grande leque de usos, como por exemplo:

1. dar o comprimento de uma lista;
2. contar o número de ocorrências de um determinado elemento;
3. incrementar um determinado valor;
4. etc.

```
tamanho :: [Char] -> Int -> Int
tamanho [] a = a
tamanho (x:xs) a = tamanho xs a+1
```

O exemplo dado, dá o comprimento de uma determinada lista, através do incremento do valor a. A particularidade dos contadores é que é necessário passar o seu valor inicial no momento da invocação da função.

Funções de ordem Superior

Funções de ordem superior: é a capacidade que uma determinada linguagem tem de permitir a referência a funções, que trabalham sobre outras funções do mesmo tipo. As funções *map*, *foldr/foldl*, *filter*, entre outras, são exemplos de *funções de ordem superior*.

Map

A função map, aplica uma determinada operação a todos os elementos de uma dada lista, criando assim uma nova lista.

```
soma :: [Int] -> [Int]
soma [] = []
soma x = map (5+) x
```

A função map, no caso apresentado, soma o valor 5 a todos os elementos da lista x.
Nota: Ao contrário das funções recursivas, no uso de funções de ordem superior não é necessário a divisão entre a cabeça e a cauda da lista.

O resultado da aplicação desta função á lista [1,2,3,4,5] é:

```
[6,7,8,9,10]
```

Filter

A função *filter* utiliza um predicado para obter resultados. Deste modo recebe uma lista e um "teste", os valores que passarem dão origem à lista de resultados.

```
par :: [Int] -> [Int]
par [] = []
par x = filtar (verifica) x

verifica :: Int -> Bool
verifica x | (mod x 2)==0 = True
           | otherwise = False
```

No caso apresentado, a função filter é usada para criar a lista de números pares. Isto acontece com recurso a uma função auxiliar (verifica) que retorna um Booleano consoante o caso.

Fold

-foldl

A função foldl aplica uma determinada operação a uma lista completa da esquerda para a direita. Deste modo se for necessário somar todos os elementos de uma lista, esta é a função a utilizar. No entanto ela tem a particularidade de se poder acrescentar mais um elemento à operação.

```
somaLis :: [Int] -> Int
somaLis [] = []
somaLis x = foldl (h) t x
```

Assim se se quiser obter a soma de todos os valores da lista basta substituir t por 0, e h por +. *Nota:* esta função aplica a operação h, que pode ser +, -, /, *, a todos os elementos da lisa, e ainda a t. Sendo que este pode ser substituído por qualquer valor.

-foldr A função foldr faz a mesma operação da função foldl, excepto que neste caso as operações se realizam da direita para a esquerda.

```
somaLis :: [Int] -> Int
somaLis [] = []
somaLis x = foldr (h) t x
```

Assim se se quiser obter a soma de todos os valores da lista basta substituir t por 0, e h por +.

Nota: esta função aplica a operação h, que pode ser +, -, /, *, a todos os elementos da lisa, e ainda a t. Sendo que este pode ser substituído por qualquer valor.

-foldr vs foldl

Sendo assim, o que faz diferir os resultados entre a função *foldr* e *foldl*?
As duas funções, apesar de terem nomes idênticos, trabalham de maneira diferente, sendo que a função *foldr* aplica a operação a partir do último elemento da lista, e a função *foldl* a partir do primeiro elemento.

Aqui fica o exemplo da execução das duas funções, recebendo os mesmos parâmetros.

```
foldl (-) 1 [1,2,3,4] = (((1-1)-2)-3)-4 = -9
foldr (-) 1 [1,2,3,4] = 1-(2-(3-(4-1))) = -1
```

Conclusão

Ao longo deste tutorial, foram abordados diversos conceitos básicos de programação em Haskell. Apesar de não serem conhecimentos muito profundos, já são capazes de apresentar algum poder e facilidade de utilização desta linguagem.
Todos os exemplos apresentados estão funcionais e podem ser testados através da criação de um ficheiro .hs, e testados num dos interpretadores aconselhados.