Tutorial de Haskell

Introdução

O Haskell é uma linguagem de programação funcional, baseada no lambda-calculus, que surgiu nos anos 80. Esta linguagem possui algumas das mais recentes características das linguagens funcionais como *lazy evaluation*, *pattern matching* e modularidade. Trata-se de uma linguagem não muito conhecida e como tal ainda não muito usado em termos comerciais. Apesar de tudo, é uma das linguagens mais populares no meio académico e muito usada em investigação. Os programas tanto podem ser compilados (usando um compilador como o GHC) ou interpretados (usando o GHCi ou o HUGS, por exemplo). Para fazer o download basta irem a http://www.haskell.org/haskellwiki/Implementations.

Preliminares

Um programa em Haskell deve começar com a linha `module Nome_Mod where`, onde *Nome_Mod* é o nome deste módulo, que tem que começar por uma letra maiúscula. O nome do ficheiro deverá ser *Nome_Mod.hs*, embora não seja obrigatório, será necessário caso se queira importar este módulo noutro programa. Para definir um comentário de uma linha usa-se os caracteres '`--`' e para comentários de mais do que uma linhas usa-se '`{-`' para iniciar o bloco de comentários e '`-}`' para fechar.

O primeiro programa

```
module PrimeiroProg where

funcao :: Int -> Int
funcao x = x^2
```

Talvez estivessem à espera do famoso *Hello World*, no entanto definir uma função que imprime uma mensagem no ecrã é um pouco complexo (obriga a utilizar *monads*), como tal fica um exemplo da função $f(x)=x^2$.

Para testar o programa basta guardar o ficheiro (*PrimeiroProg.hs*) e, depois de instalado o GHC, executar o comando `ghci PrimeiroProg.hs` (em Windows normalmente é só clicar duas vezes sobre o ícone do ficheiro que o GHCi abre automaticamente). Para testar a função faz-se `funcao n` onde *n* é um inteiro qualquer e deverão obter como resultado n^2.

De referir que a 3ª linha (a assinatura da função) não é obrigatória, o Haskell possui um mecanismo de inferência de tipos que lhe permite determinar o tipo das funções.

Como se pode ver é muito simples definir funções matemáticas em Haskell. Apresentam-se mais alguns exemplos:

```
f1 :: Float -> Float -> Bool
f1 x y = x * y > 0

divInt :: Int -> Int -> Int
divInt a b = if b == 0 then a else div a b

resto :: Int -> Int -> Int
resto a b = if b /= 0 then a `mod` b else 0

soma x y = (+) x y

-- verifica se um caracter é uma vogal
vogal :: Char -> Bool
vogal a = if a=='a' || a=='e' || a=='i' || a=='o' || a=='u'
            then True
            else False

f2 :: (Float,Float,Float) -> Float -> Bool
f2 (a,b,c) d = sin ((a * b) / c) > d
```

Aqui pode-se ver exemplos de funções com `if` (que ao contrário de muitas linguagens, precisam sempre do `else`) e os operadores prefixos `div` e `mod`. Na função `resto` pode-se ver como *transformar* um operador prefixo (neste caso o `mod`) num infixo, basta envolve-lo por acentos graves (e não plicas). Também operadores infixos podem passar a prefixos envolvendo-os com parêntesis (tal como acontece na função `soma`).

Repare-se que na função vogal o `then` passa para a linha seguinte, isto é possível desde de que a linha esteja identada (bastava ser um caracter).

Por último pode-se ver a utilização de tuplos. Neste exemplo todos os elementos são do tipo `Float`, mas podiam ser de tipos diferentes.

Para simplificar as coisas, pode-se considerar que uma função do tipo `Int -> Int -> Int` é uma função que recebe dois inteiros e devolve um inteiro. No entanto isso não é totalmente verdade, mas isto são pormenores um pouco mais avançados da linguagem.

É agora possível testar as funções fazendo, por exemplo, `resto 14 3`, `vogal 'b'`, `f2 (1,2,3) 0`...

Recursividade

Em Haskell não há ciclos, como tal a maior parte dos problemas resolvem-se com recursividade.

```
factorial n = if n == 0 then 1 else n * factorial (n-1)

quadrado n = if n == 0 then 0 else quadrado (n-1) + 2 * n - 1
```

Mais à frente serão apresentados exemplos mais complexos…

Outras formas de exprimir condicionais

As seguintes funções são todas equivalentes. Tratam-se de funções que verificam se um valor é 0 ou não.

```
isZero a = if a == 0 then True else False

isZero a | a == 0 = True
         | a /= 0 = False

isZero a | a == 0 = True
         | otherwise = False

isZero a = case a of
             0 -> True
             x -> False

isZero a = case a of
             0 -> True
             _ -> False

isZero 0 = True
isZero _ = False
```

É relativamente fácil perceber os exemplos. Talvez os mais complicados sejam os três últimos onde é usado *pattern matching*. Basicamente aqui o programa tenta *encaixar* os argumentos em padrões (neste caso verificar se coincide com 0). O caracter '_' significa *qualquer coisa*, ou seja, tudo encaixa no '_'. De referir ainda que no último exemplo não se pode trocar a ordem das linhas, caso contrário daria sempre `False`, pois o programa executa a primeira *versão* que encaixe no padrão. Nestes exemplos apenas se têm dois casos, mas se fossem mais o código seria semelhante.

Listas e strings

O Haskell suporta também um tipo de dados muito útil, as listas. Ao contrário dos tuplos, os elementos de uma lista têm que ser todos do mesmo tipo. Vejamos alguns exemplos.

```
comprimento l=length l

vazia [] = True
vazia _  = False

primeiro :: [Int] -> Int
primeiro lista = head lista

cauda [] = []
cauda lista = tail lista
```

```
junta :: [Int] -> [Int] -> [Int]
junta lista1 lista2 = lista1 ++ lista2

adiciona :: Int -> [Int] -> [Int]
adiciona x l = x : l

somatorio [] = 0
somatorio (x : xs) = x + (somatorio xs)
```

Aqui foram utilizadas as funções `head` (devolve o primeiro elemento da lista), `tail` (devolve a *cauda* da lista), '`++`' (junta duas listas) e '`:`' (adiciona um elemento no início de uma lista). A última função calcula o somatório de uma lista de números. Quando recebe uma lista vazia ('`[]`') devolve 0, se não for vazia, então a lista encaixa no padrão `x : xs`, ou seja, é um primeiro elemento (*x*) mais uma cauda (*xs*), cauda essa que pode ser uma lista vazia.

As *strings* em Haskell não são mais do que listas de caracteres.

```
juntaLetra :: Char -> String -> [Char]
juntaLetra l str = l : str

juntaNoFin l str = str ++ [l]

-- a função 'take' selecciona os n primeiro elementos de uma lista
dezLetras str = take 10 str

comprimento str = length str
```

De seguida mostram-se algumas formas de definir listas.

```
-- lista dos inteiros de 1 a 100
[1..100]

-- lista dos números pares maiores do que zero e menores do que 100
[2,4..100]

-- lista dos quadrados dos numeros inteiros de 1 a 20
[n^2 | n <- [1..20]]

-- lista [(1,0),(2,1),(3,0),...]
[(n,ispar) | n <- [1..20], ispar <- [0,1] , ((mod n 2==0 && ispar==1)
|| (odd n && ispar==0))]

-- lista de todas as combinações de elementos de [1,2,3] e de [4,5,6]
[(x,y) | x <- [1,2,3], y <- [4,5,6]]

-- lista de TODOS os números inteiros maiores do que 0
[1..]
```

Um pormenor interessante, derivado do facto do Haskell ser *lazy*, e que alguns acharão estranho, é que é possível ter listas infinitas, como no último exemplo. Quando tiverem a consola do GHCi aberta digitem `[1..]` e verão o interpretador a mostrar continuamente números (até que você digitem *Ctrl+C*).

Existem algumas funções interessantes sobre listas, como por exemplo `map`, `filter` e `foldl`. Ficam aqui alguns exemplos.

```
vezes2 n = n * 2

-- aplica a função 'vezes2' a todos o elementos de uma lista
duplica :: [Int] -> [Int]
duplica l = map vezes2 l

-- ou alternativamente
duplica l = map (* 2) l

-- filtra todos os elementos diferentes de 0 (usa a função 'isZero'
definida anteriormente)
naoZeros l = filter (not . isZero) l
```

Pode-se ver também a aplicação da composição de funções através do operador '.': `(not . isZero) n` é equivalente a `not (isZero n)`.

Sinónimos de tipos e novos tipos de dados

Em Haskell podemos definir sinónimos de tipos. Por exemplo, podemos *associar* ao nome `Par` o tipo `(Int,Int)`. Para tal usamos a declaração `type`.

```
type Par=(Int,Int)
```

Desta forma escrever `Par` ou `(Int,Int)` é a mesma coisa. No próximo exemplo as funções `f1` e `f2` são exactamente a mesma coisa.

```
f1 :: (Int,Int) -> Int
f1 (x,y) = x + y

f2 :: Par -> Int
f2 (x,y) = x + y
```

Como já foi referido, *strings* em Haskell são lista de caracteres, ou seja, foram declaradas da seguinte forma: `type String=[Char]`.

Mas podemos fazer muito mais do que definir sinónimos em Haskell. Podemos definir novos tipos através da declaração `data`. Por exemplo, quando se faz a divisão de dois números o denominador não pode ser zero, caso isso aconteça é um erro. Então vai-se definir um tipo de dados que permita representar um inteiro e situações de erro e algumas funções que usam esse tipo.

```
data ResDiv = Erro | Res Int

divisao :: Int -> Int -> ResDiv
divisao _ 0 = Erro
divisao n d = Res (div n d)

mult :: ResDiv -> ResDiv -> ResDiv
mult Erro    _       = Erro
```

```
mult _        Erro    = Erro
mult (Res x) (Res y) = x * y

-- converte um valor do tipo ResDiv para Int.
-- o valor erro é considerado 0
res2int :: ResDiv -> Int
res2int Erro    = 0
res2int (Res x) = x
```

Neste caso o tipo `ResDiv` só servia quando se estivesse a trabalhar com inteiros. Mas pode-se generalizá-lo para qualquer tipo de dados.

```
data ResDiv a = Erro | Res a

divisao :: Int -> Int -> ResDiv Int
divisao _ 0 = Erro
divisao n d = Res (div n d)

mult :: ResDiv Float -> ResDiv Float -> ResDiv Float
mult Erro     _        = Erro
mult _        Erro     = Erro
mult (Res x) (Res y) = x * y

-- converte um valor do tipo ResDiv para Int.
-- o valor erro é considerado 0
res2int :: ResDiv Double -> Double
res2int Erro    = 0
res2int (Res x) = x
```

No entanto, nem era preciso definir este tipo de dados. Já existe o tipo `Maybe`, que faz exactamente a mesma coisa. A sua definição é a seguinte:

```
data Maybe a = Just a | Nothing
```

A única diferença em relação ao tipo previamente definido são mesmo os nomes.

Mas também existem tipos recursivos. Vejamos como definir expressões matemáticas e uma função que as avalia.

```
data Exp a = Val a                -- um numero
           | Neg (Exp a)          -- o simetrico de uma expressao
           | Add (Exp a) (Exp a)  -- a soma de duas expressoes
           | Sub (Exp a) (Exp a)  -- subtraccao de duas expressoes
           | Mul (Exp a) (Exp a)  -- multiplicacao ...
           | Div (Exp a) (Exp a)  -- divisao ...

avalia (Val x)          = x
avalia (Neg exp)        = - (avalia exp)
avalia (Add exp1 exp2)  = (avalia exp1) + (avalia exp2)
avalia (Sub exp1 exp2)  = (avalia exp1) - (avalia exp2)
avalia (Mul exp1 exp2)  = (avalia exp1) * (avalia exp2)
avalia (Div exp1 exp2)  = (avalia exp1) / (avalia exp2)
```