Aula Prática 3

Paradigmas da Programação I / Programação Funcional

ESI/MCC 1^o ano (2005/2006)

Nesta aula pretende-se que os alunos: consolidem os conceitos de *expressão*, *tipo* e *redução*; compreendam a noção de *padrão* e a *concordância de padrões*; compreendam o conceito de *definição local*; explorem as *definições multi-clausais de funções*; programem com listas por compreensão.

1 Definindo funções

A definição de funções pode ser feita por um conjunto de equações da forma:

nome arg1 arg2 ... argn = expressão

Em que cada argumento da função tem que ser um *padrão*. Um padrão pode ser uma variável, uma constante, ou um "esquema" de um valor atómico (isto é, o resultado de aplicar construtores básicos dos valores a outros padrões). Além disso, estes padrões não podem ter variáveis repetidas (*padrões lineares*).

Exemplo: `5` é um padrão do tipo `Int`;
`[x,'A',y]` é um padrão do tipo `[Char]`;
`(x,8,(True,b))` é um padrão do tipo `(a,Int,(Bool,b))`.

Mas, `[x,'a',1]`, `(2,x,(z,x))` e `(4*5,y)` não podem ser padrões de nenhum tipo. Porquê ?

Quando se define uma função podemos incluir informação sobre o seu tipo. No entanto, essa informação não é obrigatória.

O tipo de cada função é *inferido automaticamente* pelo interpretador. Essa inferência tem por base o princípio de que ambos os lados da equação têm que ser do mesmo tipo. É possivel declararmos para uma função um tipo mais especifico do que o tipo inferido automáticamente.

Exemplo:
```
seg :: (Bool,Int) -> Int
seg (x,y) = y
```
Se não indicarmos o tipo `seg::(Bool,Int)->Int` qual será o tipo de `seg` ?

Podemos definir uma função recorrendo a várias equações, mas todas as equações têm que ser bem tipadas e de tipos coincidentes.

Exemplo:
```
f :: (Int,Char,Int) -> Int
f (y,'a',x) = y+x
f (z,'b',x) = z*x
f (x,y,z) = x
```

Cada equação é usada como *regra de redução* (cálculo). Quando uma função é aplicada a um argumento, a equação que é selecionada como regra de redução é a 1ª equação (a contar de cima) cujo padrão que tem como argumento *concorda* com o argumento actual (*pattern matching*).

Note que podem existir várias equações com padrões que concordam com o argumento actual. Por isso, a ordem das equações é importante, pois define uma prioridade na escolha da regra de redução.

Tarefa 1: Indique, justificando, o valor das seguintes expressões:

i) `f (3,'a',5)`

ii) `f (9,'B',0)`

iii) `f (5,'b',4)`

O que acontece se alterar a ordem das equações que definem `f` ?

Tarefa 2: Considere a seguinte função:

```
opp :: (Int,(Int,Int)) -> Int
opp z = if ((fst z) == 1)
        then (fst (snd z)) + (snd (snd z))
        else if ((fst z) == 2)
             then (fst (snd z)) - (snd (snd z))
             else 0
```

Defina uma outra versão função `opp` que tire proveito do mecanismo de *pattern matching*. Qual das versões lhe parece mais legível ?

Em Haskell é possível definir funções com alternativas usando *guardas*. Uma guarda é uma expressão booleana. Se o seu valor for `True` a equação correspondente será usada na redução (senão o interpretador tenta utilizar a equação seguinte).

Exemplo: As funções `sig1`, `sig2` e `sig3` são equivalentes. Note que `sig2` e `sig3` usam guardas. `otherwise` é equivalente a `True`.

```
sig1 x y = if x > y then 1
                    else if x < y then -1
                                  else 0

sig2 x y | x > y  = 1
         | x < y  = -1
         | x == y = 0

sig3 x y
    | x > y     = 1
    | x < y     = -1
    | otherwise = 0
```

Tarefa 3: Defina novas versões da função `opp` usando definições com guardas.

Tarefa 4: Relembre a função factorial definida na última aula. Podemos definir a mesma função declarando as duas clausulas que se seguem:

```
fact :: Int -> Int
fact 0 = 1
fact n = n * fact (n-1)
```

Esta definição de `fact` comporta-se bem sobre números naturais, mas se aplicarmos `fact` a um número negativo (o que matemáticamente não faz sentido) a função não termina (verifique). Use uma guarda na definição de `fact` para evitar essa situação.

O Haskell aceita como *padrões sobre números naturais*, expressões da forma: (*variável* `+` *número natural*). Estes padrão só concorda com números não inferiores ao número natural que está no padrão. Por exemplo, o padrão `(x+3)` concorda com qualquer inteiro maior ou igual a 3, mas não concorda com 1 ou 2. Note ainda que expressões como, por exemplo, `(n*5)`, `(x-4)` ou `(2+n)`, não são padrões. (Porquê ?)

Exemplo: Podemos escrever uma outra versão da função factorial equivalente à função que acabou de definir, do seguinte modo:

```
fact 0 = 1
fact (n+1) = (n+1) * fact n
```

Note como esta função assim declarada deixa de estar definida para números negativos.

Tarefa 5: Considere a definição matemática dos números de Fibonacci:

$$\begin{aligned} &fib(0) = 0 \\ &fib(1) = 1 \\ &fib(n) = fib(n-2) + fib(n-1) \quad \text{se } n \geq 2 \end{aligned}$$

Defina em Haskell a função de Fibonacci.

2 Definições locais

Todas as definições feitas até aqui podem ser vistas como *globais*, uma vez que elas são visíveis no módulo do programa aonde estão. Mas, muitas vezes é útil reduzir o âmbito de uma declaração. Em Haskell há duas formas de fazer *definições locais*: utilizando expressões `let...in` ou através de cláusulas `where` junto da definição equacional de funções.

Exemplo: As funções `dividir1`, `dividir2` e `dividir3` são equivalentes. As declarações de `q` e `r` são apenas visíveis na expressão que está a seguir a `in`. As declarações de `quociente` e `resto` são apenas visíveis no lado direito da equação que antecede `where`. (Teste estas afirmações.)

```
dividir1 x y = (div x y, mod x y)

dividir2 x y = let q = div x y
                   r = x `mod` y
               in (q,r)

dividir3 x y = (quociente,resto)
  where quociente = x `div` y
        resto = mod x y
```

As declarações locais podem ser de funções e de identificadores (fazendo uso de padrões).

Tarefa 6: Analise e teste a função `exemplo`. Nota: `_` é uma variável anónima nova (útil para argumentos que não são utilizados).

```
exemplo y = let k = 100
                g (1,w,z) = w+z
                g (2,w,z) = w-z
                g (_,_,_) = k
            in  ((f y) + (f a) + (f b) , g (y,k,c))
  where  c = 10
         (a,b) = (3*c, f 2)
         f x = x + 7*c
```

Tarefa 7: A seguinte função calcula as raizes reais de um polínómio $a\,x^2 + b\,x + c$. Escreva outras versões desta função (por exemplo: com `let...in`, sem guardas, ...).

```
raizes :: (Double,Double,Double) -> (Double,Double)
raizes (a,b,c) = (r1,r2)
  where r1 = (-b + r) / (2*a)
        r2 = (-b - r) / (2*a)
        d = b^2 - 4*a*c
        r | d >= 0 = sqrt d
          | d < 0  = error ``raizes imaginarias''
```

Nota: `error` é uma função pré-definida que permite indicar a mensagem de erro devolvida pelo interpretador. (Qual será o seu tipo ?)

3 Listas

Como já vimos o Haskell tem pré-definido o tipo `[a]` que é o tipo das listas cujos elementos são todos do tipo `a`. Relembre que `a` é uma variável de tipo que representa um dado tipo (ainda por escolher).

Na realidade, as listas são construidas à custa de dois construtores primitivos:

- a lista vazia, `[] :: [a]`
- o construtor `(:) :: a -> [a] -> [a]`, que é um operador infixo que dado um elemento `x` de tipo `a` e uma lista `l` de tipo `[a]`, constroi uma nova lista, `x:l`, com `x` na 1ª posição seguida de `l`.

Exemplo: `[1,2,3]` é uma abreviatura de `1:(2:(3:[]))`, que é igual a `1:2:3:[]` porque `(:)` é associativa à direita. Portanto, as expressões: `[1,2,3]`, `1:[2,3]`, `1:2:[3]` e `1:2:3:[]` são todas equivalentes. (Teste esta afirmação no ghci.)

Os padrões do tipo lista são expressões envolvendo apenas os seus construtores `[]` e `(:)`, ou a representação abreviada de listas. Padrões com o construtor `(:)` terão que estar envolvidos por parentesis.

Exemplo: Uma função que testa se uma lista é vazia pode ser definida por:

```
vazia [] = True
vazia (x:xs) = False
```

Exemplo: A função que calcula o comprimento de uma lista está pré-definida no Prelude por:

```
length :: [a] -> Int
length [] = 0
length (_:t) = 1 + (length t)
```

Exemplo: Uma função que recebe uma lista de pontos no plano cartesiano e calcula a distância de cada ponto à origem, pode definida por:

```
distancias :: [(Float,Float)] -> [Float]
distancias [] = []
distancias ((x,y):xys) = (sqrt (x^2 + y^2)) : (distancias xys)
```

Tarefa 8: Defina uma versão alternativa para a função `vazia`.

Tarefa 9: A função que soma os elementos de uma lista até à 3^a posição pode ser definida da seguinte forma:

```
soma3 :: [Integer] -> Integer
soma3 [] = 0
soma3 (x:y:z:t) = x+y+z
soma3 (x:y:t) = x+y
soma3 (x:t) = x
```

Em `soma3` a ordem das equações é importante? Porquê? Será que obtemos a mesma função se alterarmos a ordem das equações?

Defina uma função equivalente a esta usando apenas as funções pré-definidas `take` e `sum`.

Tarefa **:

i) Defina a função `transf::[a]->[a]` que faz a seguinte transformação: recebe uma lista e, caso essa lista tenha pelo menos 4 elementos, troca o 1^o com o 2^o elemento, e o último com o penúltimo elemento da lista. Caso contrário, devolve a mesma lista. Por exemplo: `transf [1,2,3,4,5,6,7]` $\Rightarrow$ `[2,1,3,4,5,7,6]`.
(*Sugestão:* utilize as funções pré-definidas `length` e `reverse`, e tire proveito das definições com guardas e das declarações locais com padrões.)

ii) Defina uma função `somaPares25::[(Int,Int)]->(Int,Int)` que recebe uma lista de pares de inteiros e calcula a soma do 2^o com o 5^o par da lista.

iii) Estas funções que definiu são *totais* ou *parciais* ?

4 Listas por compreensão

Inspirada na forma de definir conjuntos por compreensão em linguagem matemática, a linguagem Haskell tem também mecanismos para definir *listas por compreensão*. Por exemplo:

Conjuntos	Listas por compreensão em Haskell
$\{2x \mid x \in \{10,3,7,2\}\}$	`[ 2*x | x <- [10,3,7,2] ]`
$\{n \mid n \in \{9,8,-2,7,3\} \wedge 0 \leq n+2 \leq 9\}$	`[ n | n <- [9,8,-2,7,3], 0<=n+2, n+2<=9 ]`
$\{4,7,...,19\}$	`[4,7..19]`
$\{(x,y) \mid x \in \{3,4,5\} \wedge y \in \{9,10\}\}$	`[ (x,y) | x <- [3,4,5], y <- [9,10] ]`
$\{5,10,...\}$	`[5,10..]`
$\{x^3 \mid x \in I\!N \wedge par(x)\}$	`[ x^3 | x <- [0..], even x ]`

Tarefa 10: Utilizando listas por compreensão:

i) Declare uma outra versão da função `distancias`.

ii) Defina a função `ePrimo :: Integer -> Bool` que testa se um dado número é primo.

iii) Defina a função `tabuada :: Int -> [(Int,Int,Int)]` que dado um inteiro `n` produz uma lista da *tabuada dos n*. Por exemplo, a tabuada dos 5 seria: `[(5,1,5), (5,2,10), (5,3,15), ..., (5,10,50)]`.

iv) Defina a função `inverte :: String -> String` que recebe um texto e constroi o texto com cada palavra invertida. Por exemplo,

`inverte ''isto serve de exemplo''` $\Rightarrow$ `''otsi evres ed olpmexe''`.

v) Defina a função `iniciais :: String -> String` que recebe um nome e devolve uma string com as suas iniciais, separadas por pontos.

vi) Defina a lista (infinita) que representa a sequência de Fibonacci.