

Ficha Prática 2

Programação Funcional CC

LCC 1^o ano (2007/2008)

Resumo

Nesta ficha pretende-se trabalhar sobre os seguintes conceitos básicos da linguagem de programação funcional Haskell: noção de padrão e de concordância de padrões; definições multi-clausais de funções e a sua relação com a redução (cálculo) de expressões; definição de funções com guardas, definições locais. Pretende-se ainda trabalhar na definição de funções recursivas sobre listas, e na definição de tipos sinónimos.

Conteúdo

1	Definição (multi-clausal) de Funções	2
2	Definições Locais	5
3	Listas e Padrões sobre Listas	7
4	Definição de Funções Recursivas sobre Listas	9
5	Tipos Sinónimos	11

1 Definição (multi-clausal) de Funções

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

$$\text{nome } \text{arg1 } \text{arg2 } \dots \text{ argn} = \text{expressão}$$

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 possível declararmos para uma função um tipo mais específico do que o tipo inferido automaticamente.

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 seleccionada 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

1. Defina novas versões da função `opp` usando definições com guardas.
2. Relembre a função *factorial* definida na última ficha. Podemos definir a mesma função declarando as duas cláusulas 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 matematicamente 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 4

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
```

Também é possível fazer declarações locais de funções.

Tarefa 5

1. 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
```

2. A seguinte função calcula as raízes reais de um polinómio ax^2+bx+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 e Padrões sobre 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 (que ainda é uma incógnita).

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

- a lista vazia, `[] :: [a]`
- o construtor `(:) :: a -> [a] -> [a]`, que é um operador infixado que dado um elemento `x` de tipo `a` e uma lista `l` de tipo `[a]`, constrói 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 parêntesis.

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

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

Tarefa 6

1. Defina uma versão alternativa para a função `vazia`.
2. A função que soma os elementos de uma lista até à 3ª 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 7

1. 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 1º com o 2º 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] ⇒ [2,1,3,4,5,7,6]`.

(Sugestão: as funções pré-definidas `length` ou `reverse` poderão ser-lhe uteis.)

2. Defina uma função `somaPares24 :: [(Int,Int)] -> (Int,Int)` que recebe uma lista de pares de inteiros e calcula a soma do 2º com o 4º par da lista.
3. Estas funções que definiu são totais ou parciais ?

4 Definição de Funções Recursivas sobre Listas

As listas são definidas de uma forma recursiva como:

1. `[]` (a lista vazia) é uma lista;
2. Se `x :: a` (i.e., `x` é do tipo `a`) e `t :: [a]` (i.e., `t` é uma lista com elementos do tipo `a`) então `(x:t) :: [a]` (i.e., `x:t` é uma lista com elementos do tipo `a`).

Esta definição conduz a uma estratégia para definir funções sobre listas.

Exemplo: A função que calcula a soma dos elementos de uma lista pode ser definida como:

```
soma [] = 0
soma (h:t) = h + (soma t)
```

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 ser definida por:

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

Tarefa 8

Use a estratégia sugerida acima para definir as seguintes funções.

1. A função que calcula o produto de todos os elementos de uma lista de números.
2. A função que, dada uma lista e um elemento, o coloca no fim da lista.
3. A função que, dadas duas listas as concatena, i.e., calcula uma lista com os elementos da primeira lista seguidos dos da segunda lista. (Sugestão: analise o que acontece para ambos os casos da primeira lista.)

Para definirmos uma função que calcula a média dos elementos de uma lista de números podemos calcular a soma dos seus elementos e o comprimento da lista retornando depois o quociente entre estes dois valores. (Nota: a função `fromIntegral`, pré-definida, é aqui usada para fazer a conversão de um valor inteiro para real.)

```
media1 l = let s = sum l
           c = length l
           in s / (fromIntegral c)
```

Esta solução corresponde a percorrer a lista 2 vezes.

Tarefa 9

1. Defina uma função que, dada uma lista, calcula um par contendo o comprimento da lista e a soma dos seus elementos, percorrendo a lista uma única vez.
2. Usando a função anterior defina uma função que calcula a média dos elementos de uma lista.

Há no entanto funções em que é mais difícil evitar estas múltiplas travessias da lista.

Tarefa 10

1. Defina uma função que, dada uma lista, a divida em duas (retornando um par de listas) com o mesmo número de elementos (isto, é claro, se a lista original tiver um número par de elementos; no outro caso uma das listas terá mais um elemento).
2. Defina uma função que, dada uma lista e um valor, retorne um par de listas em que a primeira contem todos os elementos da lista inferiores a esse valor e a segunda lista contem todos os outros elementos.
3. Defina uma função que, dada uma lista de números, retorne a lista com os elementos que são superiores à média.

5 Tipos Sinónimos

Existe em Haskell a possibilidade de definir abreviaturas para tipos. Por exemplo, o tipo `String` está pré-definido como uma abreviatura para o tipo `[Char]`, através da declaração

```
type String = [Char]
```

Exemplo: Considere que queremos definir funções de manipulação de uma lista telefónica. Para isso resolvemos que a informação de cada entrada na lista telefónica conterá o nome, nº de telefone e endereço de e-mail. Podemos então fazer as seguintes definições:

```
type Entrada = (String, String, String)
type LTelef = [Entrada]
```

A função que calcula os endereços de email conhecidos pode ser definida como

```
emails :: LTelef -> [String]
emails [] = []
emails ((_,_,em):t) = em : (emails t)
```

Note que, uma vez que o tipo `String` é por sua vez uma abreviatura de `[Char]`, o tipo da função `emails` acima é equivalente a

```
emails :: ([[Char],[Char],[Char]]) -> [[Char]]
```

Tarefa 11

1. Construa um módulo com as definições apresentadas acima e verifique (usando o comando `:t`) o tipo da função `emails`.
2. Defina uma função que, dada uma lista telefónica, produza a lista dos endereços de email das entradas cujos números de telefone são da rede fixa (prefixo '2'). Não se esqueça de explicitar o tipo desta função.
3. Defina uma função que dada uma lista telefónica e um nome, devolva o par com o nº de telefone e o endereço de e-mail associado a esse nome, na lista telefónica.