Ficha Prática 1

Programação Funcional CC

LCC 1^o ano (2006/2007)

Resumo

Nesta ficha pretende-se trabalhar sobre os seguintes conceitos básicos da linguagem de programação funcional Haskell: valores e expressões; tipos básicos e tipos compostos; operadores pré-definidos; definição de funções simples; cálculo de expressões (passos de redução) simples; utilização de módulos; utilização de recursividade na definição de funções.

Conteúdo

1 Valores, Expressões e Tipos **2**

2 Funções: Tipos e Definição **5**

3 Importação de Módulos **9**

4 Introdução às Funções Recursivas **11**

1 Valores, Expressões e Tipos

Os *valores* ou *constantes* são as entidades básicas da linguagem Haskell. As *expressões* são obtidas combinando-se valores com *funções*, *operadores* (que são também funções) e *variáveis*, que **consideraremos na secção seguinte**. Observe-se que os valores são casos particulares de expressões.

Exemplos:

Valores
`5`
`67.9`
`True`
`'u'`
`''abcd''`

Expressões
`3.8 + 4.6`
`True && (not False)`
`((*) 4 ((+) 9 3))` (note os operadores infixos)
`8 * 5 + 4 * ((2 + 3) * 8 - 6)`
`(toLower (toUpper 'x'))`

Um conceito muito importante associado a uma expressão é o seu *tipo*. Os tipos servem para classificar entidades (de acordo com as suas características). Em Haskell escrevemos `e :: T` para dizer que a expressão **e** *é do tipo* **T** (ou **e** *tem tipo* **T**).

Exemplos:

```
5 :: Int
67.9 :: Float
True :: Bool
'u' :: Char
```

```
(3.8 + 4.6) :: Float
(True && (not False)) :: Bool
((*) 4 ((+) 9 3)) :: Int
(8 * 5 + 4 * ((2 + 3) * 8 - 6)) :: Int
(toLower (toUpper 'x')) :: Char
```

Tipos Básicos

O Haskell oferece os seguintes tipos básicos:

- **Bool** - Boleanos: `True, False`
- **Char** - Caracteres: `'a', 'x', 'R', '7', '\n', ...`
- **Int** - Inteiros de tamanho limitado: `1, -4, 23467, ...`
- **Integer** - Inteiros de tamanho ilimitado: `-6, 36, 45763456623443249, ...`
- **Float** - Números de vírgula flutuante: `3.5, -45.78, ...`
- **Double** - Números de vírgula flutuante de dupla precisão: `-45.63, 3.678, 51.2E7, ...`
- () - Unit: `()`

Tipos Compostos

Produtos Cartesianos

`(a1, a2, ..., an) :: (T1, T2, ..., Tn)`,
sendo `a1` to tipo `T1`, `a2` do tipo `T2`, ... `an` do tipo `Tn`.

Exemplos:

```
(3, 'd') :: (Int,Char)
(True, 5.7, 3) :: (Bool, Float, Int)
('k', (6, 2), False) :: (Char, (Int, Int), Bool)
```

Listas

`[a1, a2, ..., an]::[T]` todos os elementos, `ai`, da lista, são do tipo `T`.

Exemplos:

```
[3, 4, 3, 7, 8, 2, 5] :: [Int]
['r', 'c', 'e', '4', 'd'] :: [Char] (nota: ['r', 'c', 'e', '4', 'd']=``rce4d'')
[('a',5), ('d', 3), ('h', 9)] :: [(Char,Int)]
[[5,6], [3], [9,2,6], [], [1,4]] :: [[Int]]
```

Cálculo do Valor de uma Expressão

Um interpretador de Haskell (no nosso caso o ghci) usa definições de funções e operadores como *regras de cálculo*, para calcular o valor de uma expressão. Por exemplo, a expressão `8 * 5 + 4 * ((2 + 3) * 8 - 6)` é calculada pelos seguintes passos:

```
   8 * 5 + 4 * ((2 + 3) * 8 - 6) ⇒ 40 + 4 * (5 * 8 - 6)
                                 ⇒ 40 + 4 * (40 - 6)
                                 ⇒ 40 + 4 * 34
                                 ⇒ 40 + 136
                                 ⇒ 176
```

Tarefa 1

No ghci faça:

```
> :set +t
> fst (4, 'a')
> snd (4, 'a')
> fst (5.6, 3)
> :i fst
> :t fst
> :i tail
```

```
> :t tail
> tail [6,7,3,9]
> tail ‘‘sdferta’’
```

Observe o mecanismo de inferência de tipos do Haskell e o polimorfismo das funções `fst` *e* `tail`*.*

Tarefa 2

Infira o tipo, se existir, de cada uma das seguintes expressões, e avalie-a:

```
[True, (5>4), (not (’5’==’6’)), (True || (False && True))]
((tail ‘‘abcdef’’),(head ‘‘abcdef’’))
[(tail ‘‘abcdef’’),(head ‘‘abcdef’’)]
[4,5,6]++[3,5,8]
(tail [6,7])
concat [‘‘asdf’’, ‘‘bbb’’, ‘‘tyuui’’, ‘‘cccc’’]
```

2 Funções: Tipos e Definição

Em Haskell as funções são objectos com o mesmo estatuto que os valores de tipos básicos e compostos. Quer isto dizer que têm igualmente um *tipo*.

Funções

`f :: T1 -> T2` funções que recebem valores do tipo `T1` e devolvem valores do tipo `T2`.
`(f a) :: T2` aplicação da função `f` ao argumento `a` do tipo `T1`.

Exemplos:

```
toLower :: Char -> Char

not :: Bool -> Bool

ord :: Char -> Int

chr :: Int -> Char

fst :: (a, b) -> a

tail :: [a] -> [a]
```

Há funções às quais é possível associar mais do que um tipo (funções polimórficas). O Haskell recorre a variáveis de tipo (a, b, c, ...) para expressar o tipo de tais funções. Uma variável de tipo representa um tipo qualquer. Quando as funções são usadas, as variáveis de tipo são substituídas pelos tipos concretos adequados.

Um princípio fundamental de qualquer linguagem do programação com tipos é do *compatibilidade do tipo de uma função com os tipos dos seus argumentos*. A verificação desta condição antes da compilação dos programas protege-os da ocorrência de muitos erros durante a sua execução. Para ilustrar o comportamento do interpretador nesta situação, tente avaliar a seguinte expressão:

```
> tail 45
```

Funções Pré-definidas

O Haskell oferece um conjunto de funções pré-definidas (cf. o módulo Prelude). Alguns exemplos foram introduzidos anteriormente:

- operadores lógicos `&&, ||, not` ;
- operadores relacionais `>, <=, ==` ;
- operadores sobre produtos cartesianos `fst, snd` ;
- operadores sobre listas `head, tail, length, reverse, concat, ++` .

Funções Definidas pelo Programador

Mas podemos também definir novas funções. Uma função é definida por uma equação que relaciona os seus argumentos com o resultado pretendido:

```
<nomefunção> <arg1>...<argn> = <expressão>
```

Por exemplo:

```
ex a = 50 * a
funcao1 x y = x + (70*y)
```

Depois de definidas estas funções poderemos utilizá-las para construir novas expressões, que serão avaliadas de acordo com as definições das funções. Por exemplo:

```
funcao1 (ex 10) 1 ⇒ (ex 10) + (70*1)
                  ⇒ (50*10) + (70*1)
                  ⇒ 500 + (70*1)
                  ⇒ 500 + 70
                  ⇒ 570
```

O tipo de cada função é inferido automaticamente pelo interpretador; no entanto é considerado "boa prática" incluir explicitamente na definição de uma função o seu tipo.

Tomemos um segundo exemplo: uma função que recebe um par de inteiros e dá como resultado o maior deles. Esta função, a que chamaremos `maior`, poderá ser definida como se segue:

```
maior :: (Int, Int) -> Int
maior (x,y) = if (x > y) then x else y
```

Se quisermos agora definir uma função que calcule o maior de três inteiros, poderemos usar a função anterior nessa definição:

```
maiorde3 :: Int -> Int -> Int -> Int
maiorde3 x y z = (maior ((maior (x,y)), z))
```

Módulos de Código Haskell

As definições de funções não podem ser introduzidas directamente no interpretador de Haskell, devendo antes ser escritas num *ficheiro* a que chamaremos um *módulo*. Um *módulo* é em geral um ficheiro contendo um conjunto de definições (declarações de tipos, de funções, de classes, ...) que serão *lidas* pelo interpretador, e depois utilizadas pelo mesmo.

Um módulo Haskell é armazenado num ficheiro com extensão `.hs`, `<nome>.hs`, em que `<nome>` representa o nome do módulo, que terá que ser declarado na primeira linha do ficheiro. Por exemplo, o ficheiro `Teste.hs` deverá começar com a declaração seguinte:

```
module Teste where
   ....
```

Um exemplo de um módulo contendo duas das funções acima definidas será:

```
module Teste where

funcao1 x y = x + (70*y)
ex a = 50 * a
```

Tarefa 3

1. *Crie um ficheiro com o módulo acima apresentado.*
2. *No* `ghci` *carregue este módulo, escrevendo*

   ```
   > :l Teste.hs
   ```

3. *Use o interpretador* `ghci` *para verificar qual o tipo das funções definidas no módulo.*
4. *Avalie depois a expressão*

   `funcao1 (ex 10) 1`.

Tarefa 4

Defina e teste as seguintes funções (sugestão: crie um módulo com o nome Ficha1 para incluir as definições que efectuar nesta aula).

1. *Defina uma função que receba dois pares de inteiros e retorne um par de inteiros, sendo o primeiro elemento do par resultado a soma dos primeiros elementos dos pares de entrada, e o segundo elemento do par, o produto dos segundos elementos dos pares de entrada.*
2. *Escreva uma função que, dados três números inteiros, retorne um par contendo no primeiro elemento o maior dos números, e no segundo elemento o segundo maior dos números.*
3. *Escreva uma função que receba um triplo de números inteiros e retorne um triplo em que os mesmos números estão ordenados por ordem decrescente.*
4. *Os lados de qualquer triângulo respeitam a seguinte restrição:* a soma dos comprimentos de quaisquer dois lados, é superior ao comprimento do terceiro lado. *Escreva uma função que receba o comprimento de três segmentos de recta e retorne um valor booleano indicando se satisfazem esta restrição.*

5. *Escreva uma função* `abrev` *que receba uma string contendo nome de uma pessoa e retorne uma string com o primeiro nome e apelido*[1]
   (e.g. `(abrev ‘‘Joao Carlos Martins Sarmento’’)=’’Joao Sarmento’’`*)*

   As funções, pré-definidas, `words` *e* `unwords` *poderão ser-lhe uteis*

   - `words :: String -> [String]`*, dá como resultado a lista das palavras (strings) de um texto (uma string)*
   - `unwords :: [String] -> String`*, constroi um texto (uma string) a partir de uma lista de palavras (strings).*

[1] Considere que o apelido só tem um nome.

3 Importação de Módulos

Um programa em Haskell é formado por um conjunto de módulos. As definições de cada módulo podem ser utilizadas internamente, ou exportadas para ser utilizadas noutros módulos.

Para utilizar definições contidas num outro módulo é necessário importá-lo explicitamente. Este tipo de ligação entre módulos estabelece-se utilizando uma declaração `import`. Como exemplo, vejamos como podemos ter acesso às funções de manipulação de caracteres e *strings* (listas de caracteres) disponíveis no módulo Char.

```
module Conv1 where

import Char

con = toLower 'A'
fun x = toUpper x
```

Uma excepção a esta regra é o módulo `Prelude`, que constitui a base da linguagem Haskell, e cujas definições estão sempre disponíveis em todos os outros módulos, sem que seja necessário importá-lo.

Tarefa 5

1. *Crie um ficheiro com o módulo acima apresentado. Use o interpretador* `ghci` *para experimentar a função* `fun` *e ver o valor da constante* `con`*.*

2. *Crie um ficheiro* `Exemp.hs` *com o módulo seguinte:*

   ```
   module Exemp where
   import Conv2
   import Char

   conv x = if (isAlpha x) then (upperandlower x)
               else []
   ```

   e outro ficheiro com o módulo `Conv2`*:*

   ```
   module Conv2 where
   import Char

   upperandlower c = [(toLower c), (toUpper c)]
   ```

   Carregue o ficheiro `Exemp.hs` *no* `ghci` *e experimente as funções* `conv` *e* `upperandlower`*. Verifique qual o tipo das funções.*

Tarefa 6

Consulte as definições oferecidas pelo módulo `Char`*, escrevendo*

```
> :b Char
```

4 Introdução às Funções Recursivas

Considere agora a seguinte definição matemática da função factorial para números inteiros não negativos:

```
0! = 1
n! = n*(n-1)*(n-2)*...*1
```

Notando que `n! = n*(n-1)!`, uma possivel definição em Haskell da função factorial, será:

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

Repare que esta função é recursiva, i.e. ela aparece na própria expressão que a define. Diz-se também que a função *se invoca a si própria*. O cálculo da função termina porque se atinge sempre o *caso de paragem* (n=0).

Tarefa 7

1. *Defina uma função que calcule o resultado da exponenciação inteira* x^y *sem recorrer a funções pré-definidas.*

2. *Defina uma função que recebe uma lista constrói o par com o primeiro e o último elemento da lista.*

3. *Defina uma função que dada uma lista dá o par com essa lista e com o seu comprimento.*

4. *Defina uma função que dada uma lista de números calcula a sua média.*