Aula Prática 2

Paradigmas da Programação I / Programação Funcional

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

Nesta aula pretende-se introduzir os conceitos seguintes: valores e expressões; tipos básicos e tipos compostos; operadores pré-definidos; definição de funções simples; cálculo de expressões (passos de redução) simples.

1 Programas em Haskell

Um programa em Haskell é formado por um conjunto de módulos. Cada módulo contém um conjunto de definições (declarações de tipos, de funções, de classes, ...) que podem ser utilizados internamente, ou exportados para serem utilizados noutros módulos.

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

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

Para utilizar definições contidas num outro módulo é necessário referencia-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
conv1 = putStrLn [(toLower 'A')]
```

Uma excepção a esta regra é o módulo `Prelude`, que constitui a base da linguagem Haskell, e cujas definições estão disponíveis por defeito.

Tarefa 1

1. *Crie um ficheiro com o módulo acima apresentado. Use o interpretador* `ghci` *para experimentar a função* `conv1`.

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

```
module Exemp where
import Conv2
import Char

conv2 x = if (isAlpha x) then (upperandlower x)
            else []

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

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` *(*`> :l Exemp.hs`*) e experimente as funções* `conv2` *e* `upperandlower`*. Verifique qual o tipo das funções (e.g. no ghci faça* `> :t upperandlower`*)*

Tarefa 2

Consulte as definições oferecidas pelo módulo Char (e.g. faça o comando `> :b Char`*).*

2 Valores, expressões e tipos

Os valores são as entidades básicas da linguagem Haskell. As expressões são obtidas aplicando funções a valores ou a outras expressões.
Exemplos:

`5`	`3.8 + 4.6`
`67.9`	`True && (not False)`
`True`	`((*) 4 ((+) 9 3))` (note os operadores infixos)
`'u'`	`8 * 5 + 4 * ((2 + 3) * 8 - 6)`
`` ``abcd'' ```	`(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                    (3.8 + 4.6) :: Float
67.9 :: Float               (True && (not False)) :: Bool
True :: Bool                ((*) 4 ((+) 9 3)) :: Int
'u' :: Char                 (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 virgula flutuante: `3.5, -45.78, ...`
- **Double** - Números de virgula flutuante de dupla precisão: `-45.63, 3.678, 51.2E7, ...`
- **()** - Unit: `()`

Tipos compostos

Produtos Cartesianos `(T1, T2, ..., Tn)`

`(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 `[T]`

`[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]]
```

Funções `T1 -> T2`

`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:

Nome e tipo da função	Exemplos de aplicação
`toLower :: Char -> Char`	`(toLower 'K')`
`not :: Bool -> Bool`	`(not (5 > 4))`, `(not False)`
`ord :: Char -> Int`	`(ord 'a') - (ord 'A')`
`chr :: Int -> Char`	`(chr (ord 'a'))`, `(chr (ord 'B' + ((ord 'a')-(ord 'A')))`
`fst :: (a, b) -> a`	`(fst ('h', 9.3))`, `(fst (5, 't'))`
`tail :: [a] -> [a]`	`(tail [5,6,2,9])`, `(tail ['a','b','c'])`

O tipo de cada função é inferido automáticamente. Mas 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 substituidas pelos tipos concretos adequados

Tarefa 3

No ghci faça:

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

Infira o tipo, se existir, de cada uma das seguintes expressões:

```
[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'']
```

2.1 Cálculo de expressões

O interpretador usa as definições que tem no programa como *regras de cálculo*, para simplificar (calcular) o valor de uma expressão. Por exemplo, a expressão `fun (ex 10) 1` é calculada da seguinte forma:

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

3 Definição de funções

O Haskell oferece um conjunto de funções pré-definidas (c.f. 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, concat, ++`.

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>`.
Considere como exemplo uma função que recebe um par de inteiros e dá como resultado o maior deles:

```
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 defini-la, usando a função anterior:

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

Tarefa 5

Defina as seguintes funções:

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’’` *)*

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. invoca-se a ela própria. O cálculo da função termina porque se atinge o caso de paragem (n=0).

Tarefa 6

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

2. *Defina uma função que dada uma lista dá 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.*

5. *...*

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