# Paradigmas da Programação I
# Programação Funcional
# 2004/2005
# Ficha Prática N° 2

JMV, JBB, MBB, MJF, OMP,JSP

Outubro de 2004

## Objectivos

- Compreender a estrutura de um programa Haskell e a utilização de módulos.
- Consolidar os conceitos de *expressão* e *tipo*.
- Explorar as diferentes formas de definir *funções* em Haskell.
- Explorar o tipo *lista*, a construção de listas e a sua utilização.

## Notas

- Apresente os tipos de todas as funções que escrever.
- Para a resolução destes exercícios recomenda-se como material de estudo e consulta o livro *Fundamentos de Computação – Livro 2: Programação Funcional*, de J. M. Valença e J. B. Barros.

# 1 Programas e módulos Haskell

Um programa Haskell é formado por um conjunto de módulos. Por definição, um dos módulos que constitui o programa Haskell deve ter o nome `Main` e definir a função `main`. Esta função é o ponto de entrada no programa: é ela que é invocada quando o programa é compilado e executado.

Cada módulo contém um conjunto de definições (declarações de tipos, implementações de funções , etc) que podem ser utilizados internamente, ou exportados para serem utilizados noutros módulos.

Um módulo Haskell é geralmente armazenado num ficheiro com extensão `<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 as definições contidas num outro módulo é necessário referenciá-lo explicitamente. Este tipo de ligação entre módulos estabelece-se utilizando uma declaração `import`.

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

Como exemplo, vejamos como podemos ter acesso às funções de manipulação de caracteres e strings (listas de caracteres) disponíveis no módulo `Char`.

```
module Main where

   import Char

   main = putStrLn [(toLower 'A')]
```

A compilação de um programa Haskell faz-se com o comando `ghc`. A sintaxe de invocação do compilador é simplesmente:

```
ghc -o <nome_do_executavel> <modulo1> [<outrosmodulos>]
```

### Tarefa 1

*Copie o exemplo anterior para um ficheiro, e crie um programa executável utilizando o GHC. Retire conclusões quanto à utilização da função* `toLower`*.*

Uma outra forma de trabalhar com módulos Haskell é utilizar um interpretador, por exemplo o `ghci`. O `ghci` pode ser utilizado directamente na linha de comando ou integrado no editor *Emacs*.

Utilizando um interpretador, é possível carregar/descarregar módulos dinamicamente, e testar as funções definidas num determinado módulo de forma interactiva. O enunciado do primeiro trabalho prático da disciplina contém a informação necessária para começar a utilizar o `ghci`.

## 2 Expressões

A programação numa linguagem funcional como o Haskell baseia-se na construção de um conjunto de declarações em que se associam **nomes** a **expressões**, e na avaliação do **valor** dessas expressões. Por exemplo:

```
mHoras m = m / 60

hMin h m = h * 60 + m

mDias x = (mHoras x) / 24

y = mDias 564393
```

Uma expressão é construída a partir objectos como valores constantes, outras expressões identificadas pelos seus nomes, funções pré-definidas, etc.

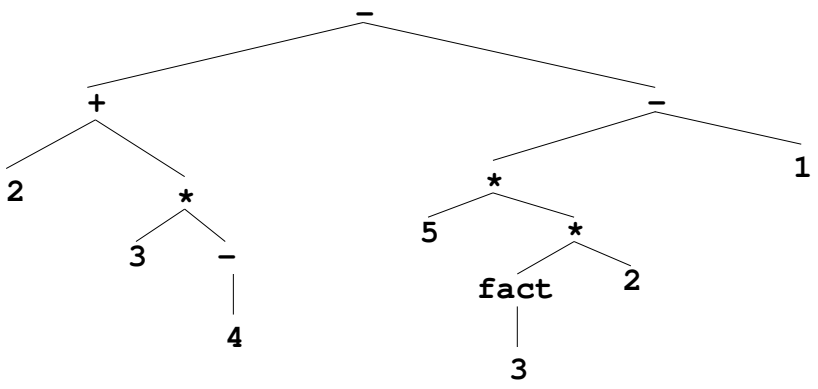


Figure 1: Árvore de expressão

Por exemplo:

```
hMin ((mDias 3) * 24)
7 + (mDias ((hMin 7 32) + (hMin 2 4)))
(mDias (hMin 3 8)) + ((mHoras 5) / 24)
mDias (hMin ((mDias 2) / 24) 45)
```

Uma forma interessante de representar expressões, pondo em evidência a sua estrutura, é através de uma árvore de expressão. Por exemplo, a expressão:

```
(2 +(3 * (-4)))-((5 * ((fact 3) * 2)) - 1)
```

pode ser representada pela árvore da Figura 1.

Note que a estrutura da árvore segue os seguinte princípio: a cada nó corresponde uma expressão. O nome da função mais exterior nessa expressão é armazenado no nó; os parâmetros dessa função aparecem, por ordem, nos ramos que dele derivam.

### Tarefa 2

*Represente a sob a forma de árvore cada uma das seguinte expressões:*

```
(2 + 3)*((4 - (5 * 3)) *( 2 - 1))
(*) ((+) 2 3) ((*) ((-) 4 ((*) 5 3)) ((-) 2 1))
(g 3 7)
(f 2 5 3)
(f (h 5) 3 (g 8 2))
```

*Sabendo que*

```
f :: Int -> Int -> Int -> Int
g :: Int -> Int -> Int
h :: Int -> Int
```

# 3 Tipos

Um conceito muito importante associado a uma expressão é o seu **tipo**. Por exemplo, sabendo que o símbolo `::` pode ser lido como *"é do tipo"*, podemos dizer que:

```
True :: Boolean
(4,5) :: (Int,Int)
'a' :: Char
toInt :: Float -> Int
mHoras :: Float -> Float
mHoras 40 :: Float
hMin :: Float -> Float -> Float
```

Nesta fase inicial, trabalharemos apenas com os tipos primitivos do Haskell (`Int`, `Integer`, `Float`, `Double`, `Char` e `Bool`), e com tuplos e listas construídos com base nesses tipos.

Os caracteres `[` e `]` são utilizados para construir listas de tamanho variável, de elementos de um determinado tipo. Por exemplo:

```
(1:2:[]) :: [Int]
(1 : [2]) :: [Int]
(7 : [2..5]) :: [Int]
[1,-1,3,17] :: [Int]
[1..100] :: [Int]
```

O operador `++` permite concatenar duas listas. Um tipo especial de lista é o tipo `String` que se define como `[Char]`, e cujos valores se representam como sequências de caracteres entre aspas e.g. `"ABC"`.

Os tuplos são construídos utilizando `(` e `)`, e podem combinar um número bem determinado de elementos de diversos tipos num só elemento composto. Por exemplo:

```
(4,5) :: (Int,Int)
(4,"ABC",5.4) :: (Int, [Char], Int)
```

As funções do Prelude `fst` e `snd` permitem obter os elementos de um par ordenado.

### Tarefa 3

*Indique, se existir e justificando, o valor e o tipo das seguintes expressões:*

```
(+) 50 80
[ True, 5]
(mHoras 3, [not True, False])
(mDias (1000 * (toInt (0.5 + 3.5))))
```

# 4 Avaliação de expressões

Um aspecto muito importante da programação em Haskell, é a forma como as expressões são avaliadas.

O Haskell caracteriza-se por utilizar *Lazy Evaluation* (retardada ou não estrita). Isto tem algumas consequências muito importantes:

- as expressões são avaliadas utilizando a estratégia *outermost/leftmost*, i.e. começando pela sub-expressão mais exterior (acima) na árvore da expressão e, no caso de existirem duas ou mais sub-expressões para serem avaliadas no mesmo nível da árvore, é escolhida aquela mais à esquerda.

- as sub-expressões só são avaliadas quando o seu valor é necessário para a avaliação da expressão que a contém (*avaliação retardada*).

- é possível construir expressões bem definidas contendo sub-expressões de valor indefinido (*avaliação não estrita*) [1].

No caso concreto da árvore da Figura 1, podemos dizer que a estratégia *outermost/leftmost* faz com que a avaliação seja realizada da seguinte forma:

1. A avaliação da raiz tem como consequência a avaliação da adição e subtracção que dela descendem, por esta ordem.
2. A avaliação da adição leva à realização dos cálculos $(-)4 = -4$, $3 * (-4) = -12$, e $2 + (-12) = -10$.
3. Por sua vez, a avaliação da subtracção leva a que se calcule $fact(3) = 6$, seguido de $6 * 2 = 12$, $5 * 12 = 60$ e $60 - 1 = 59$.
4. Finalmente, calcula-se a subtracção na raiz $-10 - 59 = 69$

Vejamos um outro exemplo muito simples que exemplifica o efeito de uma avaliação retardada e não estrita:

```
fst (5,2/0)
```

Esta expressão retorna o valor 5, ainda que o segundo componente do par seja um valor indefinido.

### Tarefa 4

*Para as árvores que desenhou na Tarefa 3, determine a forma como cada expressão é avaliada*

# 5 Funções

A função factorial é definida matemáticamente da seguinte forma para números inteiros não negativos:

$$\begin{cases} 0! = 1 \\ n! = n * (n-1) * (n-2) * \ldots * 1 \end{cases} \quad (1)$$

Notando que $n! = n * ((n-1)!)$, a função factorial pode ser implementada em Haskell das seguintes formas alternativas:

[1] As sub-expressões de valor indefinido têm, no entanto, que ser bem formadas, i.e. ter tipo adequado, dado que a inferência de tipos é estática.

```
-- Forma 1-- Utilizando "pattern matching"
fact :: Int -> Int
fact 0 = 1
fact n = n*(fact (n-1))

-- Forma 2 -- Utilizando if,then,else
fact :: Int -> Int
fact n = if (n==0)
     then 1
     else n*fact(n-1)

-- Forma 3  -- Utilizando "guardas"
fact :: Int -> Int
fact n  | (n==0) = 1
        | otherwise = n*fact(n-1)

-- Forma 4 -- Utilizando expressões lambda
fact :: Int -> Int
fact = \n -> if (n==0) then 1 else (n*fact(n-1))
```

Note a forma como o tipo da função é declarado antes da sua definição. Apesar de opcionais, as declarações dos tipos das funções são importantes na validação do raciocínio, e na depuração do código.

Repare que a avaliação de 5! se processará da seguinte forma:

```
fact 5
= 5 * (fact 4)
= 5 * 4 * (fact 3)
= 5 * 4 * 3 * (fact 2)
= 5 * 4 * 3 * 2 * (fact 1)
= 5 * 4 * 3 * 2 * 1 * (fact 0)
= 5 * 4 * 3 * 2 * 1 * 1
= 5 * 4 * 3 * 2 * 1
= 5 * 4 * 3 * 2
= 5 * 4 * 6
= 5 * 24
= 120
```

### Tarefa 5

*Os números de Fibbonacci definem-se da seguinte forma:*

$$\begin{cases} fib(0) = fib(1) = 1 \\ fib(n) = fib(n-1) + fib(n-2), n \geq 2 \end{cases} \quad (2)$$

*Defina quatro versões da função* `fib` *em Haskell, de acordo com as formas apresentadas atrás para a função factorial. Apresente uma traçagem da execução de uma delas no cálculo de* `fib 5`*.*

### Tarefa 6

1. *Escreva uma função que, dados quatro números inteiros, retorne a sua média aritmética.*

2. *Escreva uma função que, dados quatro números inteiros, retorne o maior deles.*

3. *Os lados de qualquer triângulo respeitam a seguinte restrição:* "a soma dos comprimentos de quaisquer dois lados, é superior ao comprimento do terceiro". *Escreva uma função que receba o comprimento de três segmentos de recta, e retorne um valor boleano indicando se satisfazem esta restrição.*

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

5. *Escreva uma função* `repetefrase` *que receba um inteiro* `n` *e um* string `s`*, e retorne um* string *que consista na repetição do* string *original* `n` *vezes.*

# 6 Definições locais

A definição de funções em Haskell rapidamente evolui para a manipulação de expressões muito complexas. Uma forma de tornar mais legíveis essas expressões é a atribuição de nomes a sub-expressões, nomes esses que são apenas válidos localmente.
O Haskell disponibiliza duas construções que permitem efectuar este tipo de definições locais: a construção `where` e a construção `let...in`.

Vejamos como exemplo a seguinte função que calcula o resultado da divisão inteira $x/y$. Note que o resultado desta função é um par ordenado contendo o quociente e o resto da divisão.

```
divInt :: Int -> Int -> (Int,Int)
divInt x y = let
   a = div x y
   b = x `mod` y
   in (a,b)

divInt:: Int -> Int -> (Int,Int)
divInt x y = (a,b) where
   a = div x y
   b = x `mod` y
```

### Tarefa 7

1. *Construa uma versão alternativa da função* `divInt` *sem utilizar as funções pré-definidas* `div` *e* `mod`*.*

2. *Escreva uma função que, dados três números inteiros, retorne um par contendo no primeiro elemento o maior deles, e no segundo elemento o segundo maior.*

3. *Escreva uma função que receba um triplo de números inteiros, e retorne um triplo em que os mesmos números inteiros estão ordenados por ordem decrescente.*

# 7 Funções sobre listas

As funções que trabalham sobre listas são geralmente recursivas porque apenas é possível aceder directamente ao primeiro elemento. Assim, para alcançar qualquer elemento que não o primeiro é necessário passar por todos os que o antecedem. Por exemplo, a seguinte função devolve o último elemento de uma lista não vazia:

```
f :: [a] -> a
f [x] = x
f (x:l) = f l
```

## Tarefa 8

1. *Defina uma função que, dada uma lista de inteiros, retorne o número de elementos de valor superior a 10.*

2. *Defina uma função que, dada uma lista de inteiros, devolva outra lista, contendo apenas os valores superiores a 10.*

3. *Defina uma função que, dadas duas listas de inteiros, retorne uma lista contendo todos os elementos das listas recebidas, pela mesma ordem, mas alternando entre elementos da primeira e segunda lista.*

4. *Escreva uma função* `repetaletras` *que receba um inteiro* `n` *e um* string `s`*, e retorne um* string *que consista nas letras de* `s`*, pela mesma ordem, mas repetidas cada uma delas* `n` *vezes.*

5. *Escreva uma função que receba um* string `s` *e retorne o mesmo* string*, alterando para maiúscula a primeira letra de cada palavra, e para minúsculas as restantes.*

6. *Escreva uma função* `mult_lista` *que receba duas listas de inteiros, e produza uma lista de listas, em que cada uma delas corresponde à multiplicação de um elemento da primeira lista por todos os elementos da segunda.*

# Anexo A - Algumas funções sobre listas definidas no módulo Prelude

```
head :: [a] -> a      -- First element.
last :: [a] -> a      -- Last element.
tail :: [a] -> [a]    -- All but first element.
init :: [a] -> [a]    -- All but last element.
null :: [a] -> Bool   -- True if list is empty.
length :: [a] -> Int    -- Returns the length of a finite list as an Int.
take :: Int -> [a] -> [a]   -- The prefix of xs of length n
drop :: Int -> [a] -> [a]   -- The suffix of xs after the first n elements
                            -- or [] if n > length xs
takeWhile :: (a -> Bool) -> [a] -> [a]  -- The longest prefix (possibly empty)
                                        -- of xs of elements that satisfy p
dropWhile :: (a -> Bool) -> [a] -> [a]  -- The remaining suffix
lines :: String -> [String]  -- Breaks a string up into a list of strings
                             --  at newline characters
words :: String -> [String]  -- Breaks a string up into a list of words
unlines :: [String] -> String -- Inverse of lines
unwords :: [String] -> String -- Inverse of words
concat :: [String] -> String -- Glues a list of strings together
reverse :: [a] -> [a] -- Returns the elements of xs in reverse order.
sum, product :: (Num a) => [a] -> a -- Compute the sum or product of a finite
                                    -- list of numbers.
zip :: [a] -> [b] -> [(a,b)]      --  Takes two lists and returns a list
                                  -- of corresponding pairs
zip3 :: [a] -> [b] -> [c] -> [(a,b,c)]  -- Takes three lists and returns a list
                                        -- of triples
unzip :: [(a,b)] -> ([a],[b])         --  Transforms a list of pairs
                                      -- into a pair of lists.
unzip3 :: [(a,b,c)] -> ([a],[b],[c])  -- Transforms a list of triples
                                      -- into a triple of lists.
```