Programação Funcional

Caderno de Exercícios

Maria João Frade & Jorge Sousa Pinto
Departamento de Informática
Universidade do Minho
2006

1º Ano LCC (2006/07)

Conteúdo

1 Ficha Prática 1

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.

1.1 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'']
```

1.2 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).*

1.3 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:*

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

```
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
```

1.4 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.*

2 Ficha Prática 2

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.

2.1 Definição (multi-clausal) de 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

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 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 4

Considere a definição matemática dos números de Fibonacci:

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

Defina em Haskell a função de Fibonacci.

2.2 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 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 ?)*

2.3 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 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
```

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 *?*

2.4 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 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.*

2.5 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.*

3 Ficha Prática 3

Nesta aula pretende-se por um lado trabalhar sobre padrões de funções recursivas sobre listas (mapeamento, filtragem, e *folding*), e por outro utilizar funções de ordem superior (`map`, `filter` e `foldr`) para definir de forma mais expedita essas mesmas funções. Finalmente, trabalhar-se-á sobre a definição de outras funções de ordem superior.

3.1 Funções de Mapeamento, Filtragem, e *Folding* sobre Listas

Certas funções recursivas sobre listas seguem padrões rígidos, o que permite classificá-las nas seguintes três categorias:

1. *Mapeamento*: aplicam a todos os elementos da lista argumento uma mesma função, obtendo-se como resultado uma lista com a mesma dimensão. Por exemplo:

```
dobros :: [Int] -> [Int]
dobros []     = []
dobros (x:xs) = (2*x):(dobros xs)

impares :: [Int] -> [Bool]
impares []     = []
impares (x:xs) = (odd x):(impares xs)
```

2. *Filtragem*: calculam como resultado uma sub-sequência da lista argumento, contendo (pela mesma ordem) apenas os elementos que satisfazem um determinado critério. Por exemplo:

```
umaouduasletras :: [String] -> [String]
umaouduasletras [] = []
umaouduasletras (x:xs)
      | (length x)<= 2 = x:(umaouduasletras xs)
      | otherwise      = umaouduasletras xs

filtra_impares :: [Int] -> [Bool]
filtra_impares [] = []
filtra_impares (x:xs)
      | odd x     = x:(filtra_impares xs)
      | otherwise = filtra_impares xs
```

3. *Folding*: Combinam através de uma operação binária todos os elementos da lista. Mais exactamente, *iteram* uma operação binária sobre uma lista, o que corresponde às bem conhecidas operações matemáticas de "somatório" ou "produtório" sobre conjuntos. Para a lista vazia resulta um qualquer valor constante (tipicamente o elemento neutro da operação binária, mas pode ser outro qualquer valor). Exemplos:

```
somaLista :: [Int] -> Int
somaLista []     = 0
somaLista (x:xs) = x+(somaLista xs)

multLista :: [Int] -> Int
multLista []     = 1
multLista (x:xs) = x*(multLista xs)
```

Tarefa 1

Defina as seguintes três novas funções, e diga se correspondem a algum dos padrões acima referidos.

1. *Função* `paresord` *que recebe uma lista de pares de números e devolve apenas os pares em que a primeira componente é inferior à segunda.*
2. *Funcção* `myconcat` *que recebe uma lista de* strings *e as junta (concatena) numa única string.*
3. *Função* `maximos` *que recebe uma lista de pares de números (de tipo* `float`*) e calcula uma lista contendo em cada posição o maior elemento de cada par.*

3.2 As Funções `map` e `filter`

Para possibilitar a definição fácil de funções que seguem os padrões anteriormente mencionados, é possível captar-se esses padrões em funções recursivas mais abstractas. Para o caso do mapeamento e da filtragem, temos as duas seguintes funções (pré-definidas):

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = (f x) : (map f xs)

filter :: (a -> Bool) -> [a] -> [a]
filter p []     = []
filter p (x:xs)
    | p x       = x:(filter p xs)
    | otherwise = filter p xs
```

Observe-se que estas funções, ditas de *ordem superior, recebem como argumento uma outra função*, que especifica, no caso de `map`, qual a operação a aplicar a cada elemento da lista, e no caso de `filter`, qual o critério de filtragem (dado por uma função de teste, ou *predicado*). Assim,

- `(map f l)` aplica a função `f` a todos os elementos da lista `l`. Observe a concordância de tipos entre os elementos da lista `l` e o dominio da função `f`.
- `(filter p l)` seleciona/filtra da lista `l` os elementos que satisfazem o predicado `p`. Observe a concordância de tipos entre os elementos da lista `l` e o domínio do predicado `p`.

A utilização destas funções permite a definição de funções *implicitamente recursivas*. Por exemplo, uma nova versão da função `dobros` pode ser definida como:

```
dobros' l = map (2*) l
```

ou de forma ainda mais simples, `dobros' = map (2*)`.

Tarefa 2

Avalie cada uma das seguintes expressões

1. `map odd [1,2,3,4,5]`
2. `filter odd [1,2,3,4,5]`
3. `map (\x-> div x 3) [5,6,23,3]`
4. `filter (\y-> (mod y 3 == 0)) [5,6,23,3]`
5. `filter (7<) [1,3..15]`

6. `map (7:) [[2,3],[1,5,3]]`

7. `map (:[]) [1..5]`

8. `map succ (filter odd [1..20])`

9. `filter odd (map succ [1..20])`

Tarefa 3

Defina novas versões de todas as funções de mapeamento e filtragem definidas na secção anterior, utilizando para isso as funções de ordem superior acima referidas.

Tarefa 4

Considere a função seguinte

```
indicativo  :: String -> [String] -> [String]
indicativo ind telefs = filter (concorda ind) telefs
  where concorda :: String -> String -> Bool
        concorda [] _ = True
        concorda (x:xs) (y:ys) = (x==y) && (concorda xs ys)
        concorda (x:xs) [] = False
```

que recebe uma lista de algarismos com um indicativo, uma lista de listas de algarismos representando números de telefone, e seleciona os números que começam com o indicativo dado. Por exemplo:
`indicativo "253" ["253116787","213448023","253119905"]`
devolve `["253116787","253119905"]`.

Redefina esta função com recursividade explícita, isto é, evitando a utilização de ***`filter`****.*

Tarefa 5

Considere a função seguinte

```
abrev :: [String] -> [String]
abrev lnoms = map conv lnoms
   where conv :: String -> String
         conv nom = let ns = (words nom)
                    in if (length ns) > 1
                          then  (head (head ns)):(". " ++ (last ns))
                          else nom
```

que converte uma lista de nomes numa lista de abreviaturas desses nomes, da seguinte forma: ["João Carlos Mendes", "Ana Carla Oliveira"] em ["J. Mendes", "A. Oliveira"].

Defina agora esta função com recursividade explícita, isto é, evitando a utilização de ***`map`****.*

3.3 A Função `foldr`

A função `foldr`, tal como `map` e `filter`, permite escrever de forma expedita, sem recursividade explícita, um grande conjunto de funções (incluindo as próprias funções `map` e `filter`).

O funcionamento desta função pode ser facilmente compreendido se se considerar que os constructores `(:)` e `[]` são simplesmente substituídos pelos dois parâmetros de `foldr`. Por exemplo, recordando que `[1,2,3] == 1:(2:(3:[]))`, tem-se que

```
foldr (+) 0 [1,2,3] => 1+(2+(3+0))
foldr (*) 1 [1,2,3] => 1*(2*(3*1))
```

Isto permite definir:

```
somaLista l = foldr (+) 0 l
multLista l = foldr (*) 1 l
```

Tarefa 6

Investigue o tipo e funcionamento de cada uma das seguintes funções do Haskell:

- *concat*
- *and*
- *or*

Escreva definições destas funções usando `foldr`*.*

Tarefa 7

Considere a seguinte definição de uma função que separa uma lista em duas partes de comprimento idêntico:

```
separa [] = ([],[])
separa (h:t) = (h:r,l)
     where (l,r) = separa t
```

Escreva uma nova definição desta função recorrendo a `foldr`*.*

3.4 Outras Funções

Na resolução das tarefas 8 e 9 utilize, sempre que lhe pareça natural, as funções `map`, `filter`, e `foldr`.

Tarefa 8

Pretende-se guardar a informação sobre os resultados dos jogos de uma jornada de um campeonato de futebol na seguinte estrutura de dados:

```
type Jornada = [Jogo]
type Jogo = ((Equipa,Golos),(Equipa,Golos))
type Equipa = String
type Golos = Int
```

Defina as seguintes funções:

1. `igualj :: Jornada -> Bool`
   que verifica se nenhuma equipa joga com ela própria.
2. `semrepet :: Jornada -> Bool`
   que verifica se nenhuma equipa joga mais do que um jogo.
3. `equipas :: Jornada -> [Equipa]`
   que dá a lista das equipas que participam na jornada.
4. `empates :: Jornada -> [(Equipa,Equipa)]`
   que dá a listas dos pares de equipas que empataram na jornada.

5. `pontos :: Jornada -> [(Equipa,Int)]`

   que calcula os pontos que cada equipa obteve na jornada (venceu - 3 pontos; perdeu - 0 pontos; empatou - 1 ponto)

Tarefa 9

Uma forma de representar polinómios de uma variável é usar listas de pares (coeficiente, expoente)

```
type Pol = [(Float,Int)]
```

Note que o polinómio pode não estar simplificado. Por exemplo,

```
[(3.4,3), (2.0,4), (1.5,3), (7.1,5)]  :: Pol
```

representa o polinómio $3.4\,x^3 + 2\,x^4 + 1.5\,x^3 + 7.1\,x^5$.

1. *Defina uma função para ordenar um polinómio por ordem crescente de grau.*
2. *Defina uma função para normalizar um polinómio.*
3. *Defina uma função para somar dois polinómios nesta representação.*
4. *Defina a função de cálculo do valor de um polinómio num ponto.*
5. *Defina uma função que dado um polinómio, calcule o seu grau.*
6. *Defina uma função que calcule a derivada de um polinómio.*
7. *Defina uma função que calcule o produto de dois polinómios.*
8. *Será que podemos ter nesta representação de polinómios, monómios com expoente negativo ? As funções que definiu contemplam estes casos ?*

Tarefa 10

Considere as duas seguintes funções:

```
merge  :: (Ord a) => [a] -> [a] -> [a]
insert :: (Ord a) => a -> [a] -> [a]
```

A primeira efectua a fusão *de duas listas ordenadas de forma crescente; a segunda insere um elemento numa lista ordenada de forma crescente:*

```
merge [1,4] [2,3] => [1,2,3,4]
insert "bb" ["aa","cc"] => ["aa","bb","cc"]
```

Uma definição possível de `insert` *é*

```
insert x [] = [x]
insert x (h:t)
    | (x<=h) = x:h:t
    | otherwise = h:(insert x t)
```

1. *Escreva a função* `merge` *utilizando* `foldr` *e* `insert`.
2. *Relembre o algoritmo de ordenação* insertion sort*, implementado em Haskell por uma função:*

   ```
   isort :: (Ord a) => [a] -> [a]
   ```

   Reescreva esta função utilizando `foldr`.

4 Ficha Prática 4

Nesta ficha pretende-se trabalhar com tipos de dados indutivos. Em particular utilizar-se-á o tipo `Maybe a`, e tipos definidos pelo utilizador com mais do que um construtor. Finalmente trabalhar-se-á com tipos *recursivos* definidos pelo utilizador.

Os casos de estudo correspondem à implementação de *tabelas*, ou *funções finitas*, em Haskell, e ainda à representação de expressões aritméticas.

4.1 Implementação de uma Tabela por uma Lista

Considere que se representa informação relativa à avaliação dos alunos inscritos numa disciplina, que tem no seu sistema de avaliação uma componente teórica e uma componente prática, pelos seguintes tipos de dados. Observe-se que cada estudante pode ter ou não já obtido nota numa das componentes da disciplina (teórica ou prática), o que se representa recorrendo a tipos `Maybe`.

```
type Nome = String
type Numero = Int
type NT = Maybe Float
type NP = Maybe Float

data Regime = Ordinario | TrabEstud
   deriving (Show, Eq)

type Aluno = (Numero, Nome, Regime, NT, NP)

type Turma = [Aluno]
```

Tarefa 1

Defina as seguintes funções:

1. `pesquisaAluno :: Numero -> Turma -> Maybe Aluno` *que efectua a pesquisa de um(a) aluno(a) pelo seu número de estudante. Observe que este é um identificador apropriado, uma vez que todos os alunos têm números diferentes.*

2. `alteraNP :: Numero -> NP -> Turma -> Turma` *e*
   `alteraNT :: Numero -> NT -> Turma -> Turma`*, que permitem alterar as notas prática e teórica de um(a) estudante numa turma.*

3. `notaFinal :: Numero -> Turma -> Maybe Float` *que calcula a nota final de um aluno segundo a fórmula* $NF = 0.6NT + 0.4NP$*. Note que só é possível calcular esta nota caso o número fornecido exista na turma, e ambas as componente NT e NP atinjam a nota mínima de 9.5 valores.*

4. `trabs :: Turma -> Turma` *que filtra os alunos trabalhadores-estudantes apenas.*

5. `aprovados :: Turma -> [(Nome, Float)]` *que apresenta as notas de todos os alunos aprovados, i.e. com nota final superior ou igual a 10 valores.*

Tarefa 2

A pesquisa nesta tabela torna-se mais eficiente se se mantiver a informação ordenada. Sendo a chave de pesquisa o número de aluno, a informação dever-se-á manter ordenada segundo estre critério.

1. *Escreva uma função de inserção ordenada de novos alunos numa turma,*
   `insere :: Aluno -> Turma -> Turma`*.*

2. *Redefina a função de pesquisa da tarefa anterior, tendo em conta que a tabela é representada por uma lista ordenada.*

Tarefa 3

Pretende-se agora distinguir os alunos que frequentam a disciplina pela primeira vez dos restantes alunos. Estes últimos poderão ter uma nota prática "congelada", obtida no ano lectivo anterior, mas poderão optar por ser de novo avaliados na componente prática no ano actual, devendo ser guardadas ambas as notas. A nota prática final será a melhor das duas.

Utiliza-se para isto o seguinte tipo de dados definido pelo utilizador, com dois construtores:

```
data Aluno = Primeira (Numero, Nome, Regime, NT, NP)
           | Repetente (Numero, Nome, Regime, NT, NP, NP)
  deriving Show
```

Reescreva todas as funções das tarefas anteriores tendo em conta esta alteração no tipo `Aluno`.

4.2 Implementação de uma Tabela por uma Árvore Binária de Pesquisa

Relembre que estas árvores se caracterizam pela seguinte propriedade (invariante de ordem): o conteúdo de qualquer nó situado à esquerda de um nó X é necessariamente menor do que o conteúdo de X, que por sua vez é necessariamente menor do que o conteúdo de qualquer nó situado à sua direita. Admitindo-se a ocorrência de elementos iguais, uma destas restrições é relaxada para "menor ou igual".

Considere agora a seguinte definição de um tipo de dados polimórfico para árvores binárias:

```
data BTree a = Empty | Node a (BTree a) (BTree a)
  deriving Show
```

Para se utilizar uma árvore binária para implementar uma tabela de alunos, basta redefinir:

```
type Turma = BTree Aluno
```

Tarefa 4

Escreva as seguintes funções, cujo significado é o mesmo considerado na secção anterior desta ficha:

1. `insere :: Aluno -> Turma -> Turma`
2. `pesquisaAluno :: Numero -> Turma -> Maybe Aluno`
3. `alteraNP :: Numero -> NP -> Turma -> Turma` *e*
   `alteraNT :: Numero -> NT -> Turma -> Turma`
4. `notaFinal :: Numero -> Turma -> Maybe Float`
5. `trabs :: Turma -> [Aluno]` *(note que os alunos trabalhadores-estudantes são aqui apresentados numa lista)*
6. `aprovados :: Turma -> [(Nome, Float)]`

4.3 Árvores de Expressão

Considere os seguintes tipos de dados utilizados para a representação de expressões aritméticas por árvores binárias:

```
data OP = SOMA | SUB | PROD | DIV
  deriving (Show, Eq)
data Expr = Folha Int | Nodo OP Expr Expr
  deriving (Show, Eq)
```

Tarefa 5

1. *Escreva uma função* **`aplica`** *que aplica um operador binário a dois argumentos inteiros:*

   ```
   aplica :: OP -> Int -> Int -> Int
   ```

2. *Escreva uma função* **`avalia`** *que procede ao cálculo do valor de uma expressão:*

   ```
   avalia :: Expr -> Int
   ```

3. *Escreva finalmente uma função* **`imprime`** *que produz uma string com a representação usual de uma expressão representada por uma árvore:*

   ```
   imprime :: Expr -> String
   ```

5 Ficha Prática 5

Nesta ficha pretende-se que os alunos: consolidem os conceitos de *classe*, *instância de classe* e de *tipo qualificado*; explorem algumas das classes pré-definidas mais usadas do Haskell (por exemplo: `Eq`, `Ord`, `Num`, `Show`, `Enum`, ...) e definam novas classes.

5.1 Classes, Instâncias de Classes e Tipos Qualificados

As classes em Haskell permitem classificar tipos. Mais precisamente, uma classe estabelece um conjunto de assinaturas de funções (os *métodos* da classe) cujos tipos que são instância dessa classe devem ter definidas.

É possível que uma classe já forneça algumas funções *definidas por omissão*. Caso uma função não seja definida explicitamente numa declaração de instância, o sistema assume a definição por omissão establecida na classe. Se existir uma nova definição do método na declaração de instância, será essa definição a ser usada.

Para saber informação sobre uma determinada classe pode usar o comando do interpretador ghci `:i` *nome_da_classe* . Em anexo apresenta-se um resumo de algumas das classes pré-definidas mais utilizadas do Haskell.

Tarefa 1

Verifique qual é o tipo inferido pelo ghci (o tipo principal), para cada uma das seguintes funções (pré-definidas): `elem`, `sum` *e* `minimum`. *Justifique o tipo da função, com base no que deverá ser a sua definição.*

Tarefa 2

Considere as seguintes declarações de tipo usadas para representar as horas de um dia nos formatos usuais.

```
data Part = AM | PM
    deriving (Eq, Show)

data TIME = Local Int Int Part
          | Total Int Int
```

1. *Defina algumas constantes do tipo* `TIME`.
2. *Defina a função* `totalMinutos :: TIME -> Int` *que conta o total de minutos de uma dada hora.*
3. *Defina* `TIME` *como instância da classe* `Eq` *de forma a que a iguladade entre horas seja independente do formato em que hora está guardada.*
4. *Defina* `TIME` *como instância da classe* `Ord`.
5. *Defina* `TIME` *como instância da classe* `Show`, *de modo a que a apresentação dos termos* `(Local 10 35 AM)`, `(Local 4 20 PM)` *e* `(Total 17 30)` *seja respectivamente:* `10:35 am`, `4:20 pm` *e* `17h30m`.
6. *Defina a função* `seleciona :: TIME -> [(TIME,String)] -> [(TIME,String)]` *que recebe uma hora e uma lista de horários de cinema, e seleciona os filmes que começam depois de uma dada hora.*
7. *Declare* `TIME` *como instância da classe* `Enum`, *de forma a que* `succ` *avance o relógio 1 minuto e* `pred` *recue o relógio 1 minuto. Assuma que o sucessor de* `11:59 pm` *é* `00:00 am`. *Depois, faça o interpretador calcular o valor das seguintes expressões:* `[(Total 10 30)..(Total 10 35)]` *e* `[(Total 10 30),(Local 10 35 AM)..(Total 15 20)]`.

Tarefa 3

Considere as declarações da classe `FigFechada` *e da função* `fun` *a seguir apresentadas*

```
class FigFechada a where
    area :: a -> Float
    perimetro :: a -> Float

fun figs = filter (\fig -> (area fig) > 100) figs
```

1. *Indique, justificado, qual é o tipo inferido pelo interpretador Haskell para a função* `fun`.

2. *No plano cartesiano um rectângulo com os lados paralelos aos eixos pode ser univocamente determinado pelas coordenadas do vértice inferior esquerdo e pelos comprimentos dos lados, ou por uma diagonal dada por dois pontos. Assim, para representar esta figura geométrica, definiu-se em Haskell o seguinte tipo de dados:*

   ```
   type Ponto = (Float,Float)
   type Lado = Float
   data Rectangulo = PP Ponto Ponto
                   | PLL Ponto Lado Lado
   ```

   Declare `Rectangulo` *como instância da classe* `FigFechada`.

3. *Defina a função* `somaAreas :: [Rectangulo] -> Float` *que calcula o somatório de uma lista de rectângulos. (De preferência, utilize funções de ordem superior.)*

5.2 Algumas das classes pré-definidas do Haskell

A classe `Eq`

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    -- Minimal complete definition: (==) or (/=)
    x == y = not (x /= y)
    x /= y = not (x == y)
```

A classe `Ord`

```
data Ordering = LT | EQ | GT
                deriving (Eq, Ord, Ix, Enum, Read, Show, Bounded)

class (Eq a) => Ord a where
    compare               :: a -> a -> Ordering
    (<), (<=), (>=), (>)  :: a -> a -> Bool
    max, min              :: a -> a -> a

    -- Minimal complete definition: (<=) or compare
    -- using compare can be more efficient for complex types
    compare x y | x==y      = EQ
                | x<=y      = LT
                | otherwise = GT

    x <= y                  = compare x y /= GT
    x <  y                  = compare x y == LT
```

```
    x >= y                  = compare x y /= LT
    x >  y                  = compare x y == GT

    max x y   | x <= y      = y
              | otherwise   = x
    min x y   | x <= y      = x
              | otherwise   = y
```

A classe Num

```
class (Eq a, Show a) => Num a where
    (+), (-), (*)  :: a -> a -> a
    negate         :: a -> a
    abs, signum    :: a -> a
    fromInteger    :: Integer -> a
    -- Minimal complete definition: All, except negate or (-)
    x - y           = x + negate y
    negate x        = 0 - x
```

A classe Show

```
class Show a where
    show      :: a -> String
    showsPrec :: Int -> a -> ShowS
    showList  :: [a] -> ShowS

    -- Minimal complete definition: show or showsPrec
    show x          = showsPrec 0 x ""
    showsPrec _ x s = show x ++ s
    showList []     = showString "[]"
    showList (x:xs) = showChar '[' . shows x . showl xs
                      where showl []     = showChar ']'
                            showl (x:xs) = showChar ',' . shows x . showl xs
```

A classe Enum

```
class Enum a where
    succ, pred           :: a -> a
    toEnum               :: Int -> a
    fromEnum             :: a -> Int
    enumFrom             :: a -> [a]             -- [n..]
    enumFromThen         :: a -> a -> [a]        -- [n,m..]
    enumFromTo           :: a -> a -> [a]        -- [n..m]
    enumFromThenTo       :: a -> a -> a -> [a]   -- [n,n'..m]

    -- Minimal complete definition: toEnum, fromEnum
    succ                  = toEnum . (1+)       . fromEnum
    pred                  = toEnum . subtract 1 . fromEnum
    enumFrom x            = map toEnum [ fromEnum x ..]
    enumFromThen x y      = map toEnum [ fromEnum x, fromEnum y ..]
    enumFromTo x y        = map toEnum [ fromEnum x .. fromEnum y ]
    enumFromThenTo x y z  = map toEnum [ fromEnum x, fromEnum y .. fromEnum z ]
```

6 Ficha Prática 6

O objectivo desta ficha é a escrita de programas que envolvam IO. Concretamente, pretende-se que os alunos construam um programa "completo" com interface por menus, e manipulação de ficheiros.

O caso de estudo é uma base de dados implementada numa árvore de binária de procura. Depois do programa ser codificado, propoem-se que o código seja compilado, criando um programa executável. As funções haskell que estão descritas nesta ficha estão disponíveis no ficheiro `ficha6.hs`.

6.1 Input / Output em Haskell

Quando realizamos uma aplicação, necessitamos de "executar" operações de *entrada/saida* de dados. Estas operações escapam à identificação realizada no paradigma funcional de "execução de um programa" como o "cálculo do valor de uma expressão" — pretende-se antes especificar *acções* que devem ser realizadas numa dada sequência. Como exemplos de operações *input/output* podemos citar: ler um valor do teclado; escrever uma mensagem no ecrán; ler/escrever um ficheiro com dados.

Em *Haskell*, a integração destas operações é realizada por intermédio do *monad IO*. Podemos entender o *monad IO* como uma marca que assinala que um valor de um dado tipo foi obtido fazendo uso de operações de entrada/saida. Assim, um valor do tipo `IO Int` pode ser entendido como "um programa que realiza operações entrada/saida e retorna um valor do tipo `Int`" (ver `leInt` no exemplo apresentado abaixo, onde é retornado um valor inteiro lido do teclado). Ora, esta distinção entre valores "puros" do tipo `Int`, e os valores obtidos por intermédio de operações entrada/saida (tipo `IO Int`) coloca um problema evidente: se tivermos uma função que opere sobre inteiros (por exemplo, a função `fact` apresentada abaixo), ela não pode ser directamente aplicada a `leInt::IO Int`. Como podemos então calcular o factorial de um valor introduzido no teclado? A resposta a esta questão encontra-se nas operações que caracterizam um *Monad* (estudadas na teórica) — na prática, é preferível utilizar a notação `do` disponibilizada pela linguagem *Haskell*.

Chamemos *computação* às expressões cujo calculo envolve operações entrada/saida (i.e. expressões do tipo `IO t`, qualquer que seja o tipo `t`). A notação `do` permite definir uma computação como a sequenciação de um conjunto de computações (o valor retornado é o da última). Mas nesta sequência vamos poder aceder aos valores que vão sendo calculados. Mais precisamente:

- a seta `<-` permite-nos aceder ao valor retornado pela computação correspondente. No exemplo apresentado abaixo, na linha `l<-getLine`, temos que `getLine::IO String` (é uma função pré-definida que lê uma linha do teclado). Assim, `l` será do tipo `String` e corresponderá ao texto introduzido pelo utilizador;
- a operação `return` permite-nos embeber um valor numa computação. Ainda no exemplo apresentado, `((read l)::Int)` permite "ler" o inteiro da string `l` (a operação inversa do `show`). Assim, `return ((read l)::Int)` corresponde à computação que retorna esse valor.

```
leInt :: IO Int
leInt = do putStr "Escreva um número: "
           l <- getLine
           return ((read l)::Int)

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

prog1 :: IO ()
prog1 = do x <- leInt
           putStrLn ("o factorial de "++(show x)++" é "++(show (fact x)))
```

Estas declarações, bem como as que são apresentadas no resto desta ficha, estão dispoíveis no ficheiro `ficha6.hs`.

Vamos agora retomar o problema (das turmas) apresentado na ficha 4, para criar um programa com interface por menus.

6.2 Declaração de Tipos

Relembre da ficha prática 4 o problema de manter numa árvore binária de procura a informação sobre os alunos inscritos numa dada disciplina. Nessa ficha foram definidos os seguintes tipos de dados (acrescentamos apenas as instâncias derivadas da classe `Read` para os tipos de dados `Regime` e `BTree`).

```
type Nome = String
type Numero = Int
type NT = Maybe Float
type NP = Maybe Float

data Regime = Ordinario | TrabEstud
   deriving (Show, Eq, Read)

type Aluno = (Numero, Nome, Regime, NT, NP)

data BTree a = Empty | Node a (BTree a) (BTree a)
  deriving (Show, Read)

type Turma = BTree Aluno
```

6.3 Construção de Menus

Vamos agora usar estas declarações para construir um programa que mantém a informação acerca dos alunos. Vamos para isso usar o tipo `IO x`. Elementos deste tipo são programas que fazem algum *Input/Output* (i.e., escrevem coisas no écran, lêem do teclado, escrevem e lêem ficheiros, ...) e que dão como resultado um elemento do tipo `x`.

O primeiro destes programas vai apenas apresentar (no écran) uma lista das opções possíveis e retornar o valor escolhido pelo utilizador.

```
menu :: IO String
menu = do { putStrLn menutxt
          ; putStr "Opcao: "
          ; c <- getLine
          ; return c
          }
    where menutxt = unlines ["",
                             "Inserir Aluno .......  1",
                             "Listar Alunos .......  2",
                             "Procurar Aluno ......  3",
                             "",
                             "Sair ................  0"]
```

Vamos agora usar este programa para escrever um outro que, após perguntar qual a opção escolhida, invoca a função correspondente. Note-se que neste caso, como de umas invocações para as outras o estado (i.e., a informação dos alunos) vai mudando, este estado deve ser um parâmetro.

```
ciclo :: Turma -> IO ()
ciclo t = do { op <- menu
             ; case op of
                '1':_ -> do { t' <- insereAluno t
                            ; ciclo t'
                            }
                '2':_ -> do { listaAlunos t
                            ; ciclo t
                            }
                '3':_ -> do { procuraAluno t
                            ; ciclo t
                            }
                '0':_ -> return ()
                otherwise -> do { putStrLn "Opcao invalida"
                                ; ciclo t
                                }
             }
```

Os programas `insereAluno`, `listaAlunos`, `procuraAluno` não são mais do que as extensões para `IO` das funções de inseção, listagem e pesquisa que definiu na ficha 5. Por exemplo, para o caso da primeira,

```
insereAluno :: Turma -> IO Turma
insereAluno t
  = do { putStr "\nNumero: "; nu <- getLine;
         putStr "Nome: "; no <- getLine;
         putStr "Regime: "; re <- getLine;
         putStr "Nota Pratica: " ; np <- getLine;
         putStr "Nota Teorica: "; nt <- getLine;
         let reg = if re=="TE" then TrabEstud else Ordinario
             pra = if np=="" then Nothing else Just ((read np)::Float)
             teo = if nt=="" then Nothing else Just ((read nt)::Float)
         in return (insere ((read nu),no,reg,pra,teo) t)
       }
```

Tarefa 1

Defina os programas `listaAlunos` e `procuraAluno`.

Tarefa 2

1. *Acrescente ao menu opções para alterar as notas de um aluno, e defina as funções correspondentes.*
2. *Faça agora o importação do módulo IO e crescente ainda ao seu programa a seguite função* `main`

   ```
   import IO

   main = do { hSetBuffering stdout NoBuffering
             ; ciclo Empty
             }
   ```

   Note que o programa `main` *arranca com a base de dados vazia (ou seja, a árvore vazia). A primeira linha do* `main` *é apenas um comando para forçar a visualização imediata daquilo que é enviado para o écran.*

6.4 Manipulação de Ficheiros

Para além de escrever e ler dados no écran e do teclado, é possível consultar e escrever informação em ficheiro.

A forma mais elementar de aceder a um ficheiro é através da leitura e escrita do conteúdo do ficheiro (visto como uma única String). Para isso usam-se as funções `readFile` e `writeFile`.

Note que, como na definição dos tipos `Regime` e `BTree a` optamos por declarar instâncias derivadas das classes `Show` e `Read`, podemos usar as funções `show` e `read` para fazer a conversão entre estes tipos e o tipo `String`.

Por exemplo, para a escrita e a leitura de uma turma podemos escrever os seguintes programas:

```
writeTurma :: Turma -> IO ()
writeTurma t = do putStr "Nome do ficheiro: "
                  f <- getLine
                  writeFile f (show t)

readTurma :: IO Turma
readTurma = do putStr "Nome do ficheiro: "
               f <- getLine
               s <- readFile f
               return (read s)
```

Temos agora todos os ingredientes para estender o programa acima dando-lhe a possibilidade de guardar e ler os dados de um ficheiro (usando, entre outras as as funções `read` e `show`).

Tarefa 3

1. *Acrescente aos programas* `menu` *e* `ciclo` *os itens necessários para estas duas operações.*
2. *Experimente o programa que acabou de definir. Insira novos alunos numa turma, guarde essa turma em ficheiro, verifique se o conteúdo do ficheiro que criou é o esperado. Depois saia do programa, carregue a turma que está guardada em ficheiro, e acrescente interactivamente novos alunos. Guarde a nova turma um outro ficheiro e verifique o seu conteúdo.*

Por vezes temos que manipular ficheiros de uma forma mais elaborada.

Tarefa 4

Suponha que os Serviços Académicos formecem informação sobre os alunos inscritos a uma disciplina em ficheiros de texto com o seguinte formato:

NÚMERO REGIME NOME*, com um aluno por linha, e em que* REGIME *apenas pode ser* `ORD` *ou* `TE`*. Por exemplo,*

```
54321 ORD Ana Maria Santos Silva
33333 TE  Paulo Ferreira
44111 TE  Pedro Moura Gomes
22233 ORD Tiago Miguel de Sousa
```

Acrescente ao seu programa, uma função que permita ler o ficheiro dos alunos inscritos e carregue sua base de dados com essa informação (criando uma turma ainda sem notas).

Tarefa 5

Pretende-se agora escrever em ficheiro a pauta final da disciplina (para posteriormente se imprimir, por exemplo). Na pauta deve constar o número do aluno, o seu nome e a nota final. A nota final pode ser um valor inteiro entre 10 e 20, ou `Rep`*, indicando que o aluno está reprovado. Por exemplo,*

```
54321 Ana Maria Santos Silva  15
33333 Paulo Ferreira          Rep
44111 Pedro Moura Gomes       17
22233 Tiago Miguel de Sousa   12
```

1. *Acrescente ao seu programa uma função que permita gravar pautas em ficheiro.*

2. *Adapte a sua resposta à alinea anterior de forma a que a pauta tenha os alunos ordenados por ordem alfabética.*

6.5 Uma versão melhorada do programa

Numa árvore binária de procura, a pesquisa de informação é particularmente eficiente se essa árvore for balanceada (equilibrada). Os algoritmos de inserção que estudamos não garantem que a árvore de procura que obtemos seja balanceada. No entanto, podemos pelo menos garantir que obtemos uma árvore balaceada no momento em que carregamos a base de dados de ficheiro. Para esse fim, vamos gravar uma turma em ficheiro como uma lista ordenada de alunos e, ao recuperar a turma de ficheiro, montamos a turma na base de dados como uma árvore balanceada.

Tarefa 6

1. *Defina uma função que permita gravar uma turma num ficheiro como uma sequência ordenada (por número) de alunos.*

2. *Defina uma função que dada uma lista ordenada cria uma árvore balanceada com os elementos dessa lista.*

3. *Defina uma função que permita ler o ficheiro (com a sequência de alunos) gerado pela função da alínea 1, e construa uma árvore de procura balanceada com esses alunos.*

4. *Acrescente aos programas* `menu` *e* `ciclo` *os itens necessários para disponibilizar estas operações de gravação para ficheiro e carregamento para árvore balanceada.*

Pretende-se agora enriquecer o programa com nova funcionalidade.

Tarefa 7

1. *Defina uma função que permita remover um aluno (dado o seu número) de uma turma, e disponibilize no menu a opção de remover um aluno da turma.*

2. *Acrescente ao seu programa qualquer outra funcionalidade que lhe pareça útil.*

6.6 Compilação de um programa Haskell

Os programas que temos criado têm sido sempre *interpretados* com o auxilio do interpretador GHCI. Uma forma alternativa de "correr" o programa é *compilando-o* por forma a obter uma ficheiro executável, que pode ser invocado ao nível da shell do sistema operativo. Para isso é necessário utilizar o compilador de Haskell, GHC.

Para criar programas executáveis o compilador Haskell precisa de ter definido um módulo `Main` com uma função `main` que tem que ser de tipo IO . A função `main` é o ponto de entrada no programa, pois é ela que é invocada quando o programa compilado é executado.

A compilação de um programa Haskell, usando o Glasgow Haskell Compiler, pode ser feita executando na shell do sistema operativo o seguinte comando:

`ghc -o` *nome_do_executável* `--make` *nome_do_ficheiro_do_módulo_principal*

Tarefa 8

1. *Compile o programa de forma a dispôr de um programa executável. (Note que é necessário que o nome do módulo onde se encontra definido o programa main se chame Main.)*
2. *Abra uma shell do sistema operativo e invoque o programa executável que foi criado pela compilação.*