Aula Prática 9

Paradigmas de Programação I / Programação Funcional

ESI/MCC 1º ano (2005/2006)

O objectivo desta aula é a escrita de programas que envolvam IO.

1 Tipos

Relembre da aula prática 7 o problema de manter numa árvore binária de procura a informação sobre os alunos inscritos numa dada disciplina.
Uma forma possível de definir o tipo aluno é:

```
data Aluno = A String Int Float Float
   deriving (Show,Read)
```

Considere ainda as árvores binárias definidas como

```
data BSTree a = Vazia | Nodo a (BSTree a) (BSTree a)
   deriving (Show,Read)
```

Tarefa 1 Adapte as funções `insere`, `pesquisa` e `inorder` definidas na Aula 7 para os tipos definidos acima.

2 Menus

Vamos agora usar estas funções 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 Char
menu = do { putStrLn menutxt
          ; putStr "Opção: "
          ; c <- getChar
          ; return c
          }
    where menutxt = unlines ["Inserir Aluno ...  1",
                             "Listar Alunos ...  2",
                             "Procurar Aluno ..  3",
                             "",
                             "Saír .............  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 :: BSTree Aluno -> IO ()
ciclo a = do { o <- menu
             ;  case o of
                '1' -> do { a' <- insereAluno a
                          ; ciclo a'
                          }
                '2' -> do { listaAlunos
                          ; ciclo a
                          }
                '3' -> do { procuraAluno a
                          ; ciclo a
                          }
                '0' -> return ()
             }
```

Os programas `insereAluno`, `listaAlunos`, `procuraAluno` não são mais do que as extensões para `IO` das funções `insereNumAluno`, `inorder` e `pesquisaNumAluno`. Por exemplo, para o caso da primeira,

```
insereAluno :: BSTree Aluno -> IO (BSTree Aluno)
insereAluno a
  = do { putStr "Número: "; nu <- getLine
       ; putStr "Nome: "; no <- getLine
       ; putStr "Nota Prática: " ; p <- getLine
       ; putStr "Nota Teórica: "; t <- getLine
       ; return (insere a (A no (read nu) (read p) (read t)))
       }
```

Tarefa 2 Defina os programas `listaAlunos` e `procuraAluno`.

3 Escrita em Ficheiro

Para além de escrever e ler dados no ecrã e do teclado, é possível usar o tipo `IO ...` para 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 `Aluno` e `BSTree` optámos por usar instâncias derivadas das classes `Show` e `Read`, podemos usar as funções `show` e `read` para fazer a conversão entre estes tipos e Strings. E são estas últimas que vamos escrever e ler de um ficheiro.
Por exemplo para a leitura de uma árvore podemos escrever o seguinte programa:

```
loadBSTree :: (Read a) => IO (BSTree a)
loadBSTree = do { putStr "Nome do ficheiro: "
                ; f <- getLine
                ; bt <- readFile f
                ; return (read bt)
                }
```

Tarefa 3 Defina o correspondente programa para escrever uma árvore num ficheiro.

Temos agora todos os ingredientes para estender o programa acima dando-lhe a possibilidade de guardar e ler os dados de um ficheiro.

Tarefa 4 Acrescente aos programas `menu` e `ciclo` os itens necessários para estas duas operações. Acrescente ainda ao seu programa a seguinte função `main`.

```
main :: IO ()
main = ciclo Vazia
```

Tarefa 5 Compile o seu programa usando o comando `ghc -o xxx --make xxx.hs`. Note que é necessário que o nome do módulo onde se encontra definida o programa `main` se chame `Main`.

4 ... and now for something completely different (Monty Python, 1976)

A abordagem seguida atrás é baseada no uso das instâncias derivadas das classes `Show` e `Read`.
Nesta secção vamos definir instâncias próprias destas classes.
Comecemos pela conversão **para** String, i.e., a definição das instâncias da classe `Show`.

```
instance Show Aluno where
  show (A no nu p t)  = (take 8 ((show nu) ++ espacos))
                      ++ (take 36 (no ++ espacos))
                      ++ (take 8 ((show p) ++ espacos))
                      ++ (take 8 ((show t) ++ espacos))
                      ++ (resultado p t)

espacos = ' ':espacos
resultado p t = if (p < 9.5) || (t < 9.5) then "Rep"
                else (show (round (0.6 * t + 0.4 * p)))
```

Tarefa 6 Na definição da classe `Show` existe pré-definida uma função `showList` que converte listas. Experimente essa função neste caso.

Podemos redefinir a instância acima (definindo também a função `showList`) de forma a que ao aplicarmos a função `show` a uma lista de alunos eles apareçam um por linha (i.e., separados pelo caracter `'\n'`). Para isso basta incluir a seguinte definição:

```
  showList l  = showString ( unlines  (map show l))
```

Para a função `show` definida acima podemos definir a seguinte *inversa*.

```
readAluno s = let (nu:resto) = words s
                  (no,p,t) = separa resto
                  separa [x,y,z] = ([],x,y)
                  separa (h:t) = let (a,b,c) = separa t
                                 in (h:a, b, c)
              in (A (unwords no) (read nu) (read p) (read t))
```

Podemos usar esta função para declarar `Aluno` como uma instância da classe `Read`. Para isso basta escrever

```
instance Read Aluno where
   readsPrec _ s = [(readAluno s, "")]
```

Para conseguirmos converter uma árvore para uma String de forma a poder recuperar a árvore, é necessário que os vários elementos da árvore apareçam na String segundo uma travessia *pré-order* (**porquê?**).

Tarefa 7 Defina uma função de conversão de árvores binárias de procura para String e correspondente inversa.

```
showBSTree :: Show a => BSTree a -> String
readBSTree :: Read a => String -> BSTree a
```

Tarefa 8 Usando estas duas funções defina o tipo `BSTree Aluno` como instâncias das classes `Read` e `Show`.

Considere agora a seguinte função que, dada uma lista ordenada, constrói uma árvore binária de procura equilibrada.

```
list2BSTree :: (Ord a) => [a] -> BSTree a
list2BSTree a = l2BT (length a) a

l2BT 0 _ = Vazia
l2BT n l = let m = div n 2
               (e,(r:d)) = splitAt m l
           in Nodo r (l2BT m e) (l2BT (n-m-1) d)
```

Tarefa 9 Use a instância de `Ord` do tipo `Aluno` definido na Folha 8 para poder usar esta função para árvores de `Alunos`.

Tarefa 10 Altere as definições de `showBSTree` e `readBSTree` de forma a que a árvore de procura lida seja equilibrada. Note que como a função `list2BSTree` assume que a lista está ordenada, também vai ter de alterar a função de conversão da árvore numa lista.

Tarefa 11 Incorpore todos estes melhoramentos na definição do programa de manuseamento da informação sobre os alunos.