Ficha Prática 6

Programação Funcional CC

LCC 1^o ano (2006/2007)

Resumo

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`.

Conteúdo

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 correponderá 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.

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
```

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.*

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.*

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 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.*