Departamento de Computação
Universidade Federal de Ouro Preto
Programação Funcional
Prof. Lucília Figueiredo

Entrada e Saída em Haskell
Tutorial

Elton M. Cardoso
Lucília Figueiredo

30 de Junho de 2005

Conteúdo

Capítulo 1

E/S em Haskell

1.1 Ações e computações

Até aqui, todos os nossos programas eram funções que, dado um conjunto de argumentos, produziam como resultado um determinado valor. Chamamos tais funções de *funções puras* e dizemos que elas efetuam *computações*. Elas são puras no sentido de que espelham exatamente o conceito de função em matemática – o resultado da avaliação da expressão que constitui o corpo de uma função não depende da ordem de avaliação de suas subexpressões.

A maioria dos programas, entretanto, faz algo mais do que simplesmente computar um valor – em geral, programas também interagem como o usuário (ou ambiente). De fato, muitos programas são descritos exclusivamente pelo efeito que têm sobre o seu ambiente. Por exemplo, a característica essencial de um editor de textos é modificar arquivos armazenados no disco do computador. Chamamos de *ações* os fragmentos de programas que têm algum efeito sobre o ambiente – tal como modificar o conteúdo de arquivos – e dizemos que eles executam *interações*.

Note que, no caso de interações, a ordem em que elas são executadas influi no estado final do ambiente. Considere, por exemplo, uma seqüência de duas ações para ler dois valores inteiros, a partir do dispositivo de entrada padro, e então determinar a diferena entre esses valores. Suponha que os dois valores digitados pelo usuário sejam 4 e 3, nessa ordem. Se não garantirmos uma ordem de avaliação dessas ações, o resultado obtido poderá ser 1 ou -1, conforme o primeiro comando de leitura seja avaliado antes, ou depois, do segundo.

Programas, em geral, combinam computações e interação. Mesmo aqueles, tais como editores de texto, cujo propósito principal é executar interações, efetuam internamente diversas computações. Por exemplo, a função de pesquisa em um editor de textos efetua uma busca por um substring em uma lista de caracteres que representa o arquivo que está sendo editado.

Embora o bom estilo de programação recomende a separação clara entre componentes de programas que efetuam interação e computação, a maioria das linguagens de programação não força uma distinção clara entre esse dois tipos de componentes. Em Haskell, funções são distinguidas de ações por meio do sistema de tipos da linguagem. Ações de entrada e saída têm um tipo da forma `IO a`, para algum tipo `a`.

Descrevemos, a seguir, algumas funções básicas da linguagem para implementação de ações de entrada e saída de dados, bem como os mecanismos disponíveis para o sequenciamento da execução de ações em um programa. Abordamos apenas as operações de entrada e saída mais comuns: leitura de caracteres a partir do teclado, escrita de caracteres na tela, leitura

e escrita de caracteres em arquivos armazenados em disco. Outras formas mais complexas de interação, tais como a comunicação através de redes ou a entrada e saída de dados via interfaces gráficas, serão abordados em um curso mais avançado. Consulte o manual da linguagem (www.haskell.org/definiton) para maiores detalhes.

1.2 Entrada e Saída Padrão

Começamos com o clássico exemplo do programa *Hello Word*, que pode ser escrito, em Haskell, do seguinte modo:

```
module Main where
  main = putStrLn "Hello World!"
```

Como resultado da execução deste programa, é enviada uma mensagem ao sistema operacional, indicando que o texto passado como argumento para a função `putStrLn` deve ser impresso na tela (ou seja, no dispositivo de saída padrão).

As seguintes funções podem ser usadas para imprimir um string na tela, sendo a única diferença entre elas o fato de que a segunda inicia uma nova linha, depois de imprimir o string:

```
putStr   ::  String -> IO ()
putStrLn ::  String -> IO ()
```

O tipo de retorno dessas funções – `IO ()` – indica que é executada uma ação de E/S e nenhum valor é retornado[1]. Dizemos que funções como `putStr` e `putStrLn` são *funções de E/S* ou *funções de interação*, em contraposição a funções puras, tais como, por exemplo `(++)`.

Para ler uma linha de caracteres do teclado, podemos usar a função:

```
getLine ::  IO(String)
```

O tipo desta função indica que ela executa uma ação de entrada de dados e retorna um valor do tipo `String`.

1.3 Combinando ações

Consideramos, até agora, ações individuais, tais como `putStr` e `getLine`. Entretanto, em um programa, usualmente necessitamos combinar ações de modo a implementar uma ação mais complexa. Suponha que queremos ler uma linha do teclado e imprimir essa linha na tela. Poderíamos tentar escrever:

```
putStr getLine        -- Erro de tipo!!
```

Mas isso não funciona! Porque? A função `getLine` tem tipo `IO String` e `putStr` espera um string puro como argumento, ao invés de uma ação de E/S que produz um string como resultado. Como dissemos anteriormente, Haskell distingue entre computações puras e interações por meio de tipos. Expressões de tipo `String` e `IO String` denotam objetos diferentes e Haskell evita que eles sejam confundidos.

Quando desejamos combinar ações, devemos usar a notação `do`, que tem a forma geral:

[1] Podemos associar o tipo `()` ao tipo void de Java.

```
do
  <comando 1>
  <comando 2>
  ⋮
  <comando n>
```

onde cada comando é uma ação de E/S, elementar ou composta. Quando estamos interessados em obter o resultado de uma ação, podemos ligar o resultado dessa ação a uma variável v, usando o operador `(<-) :: IO (a) -> a`, da seguinte forma:

v `<- <ação>`

Podemos agora combinar `getLine` e `putStr` do seguinte modo:

```
do input <- getLine
   putStr input
```

Como `getLine` tem tipo `IO String`, a variável `input` denota o string lido por `getLine`, de tipo `String`.

Podemos agora definir uma função que efetua a ação composta acima, ou seja, lê um string do teclado e ecoa esse string na tela:

```
ecoLine :: IO ()
ecoLine = do input <- getLine
             putStr input
```

Exercício: Defina um programa que lê do teclado e imprime na tela, seguidamente, duas linhas de texto.

1.3.1 Definindo funções de interação

Assim como no caso de computações, é útil definir funções que executam ações mais complexas, pela combinação de ações elementares. Por exemplo, suponha que desejamos definir uma função `ask`, que dada uma pergunta (na forma de um string), exibe essa pergunta na tela, lê a resposta digitada pelo usuário e retorna a resposta lida, como um resultado do tipo `IO String`. Essa função poderia ser definida do seguinte modo:

```
ask :: String -> IO String
ask question = do putStr question
                  getLine
```

Podemos usar essa função em um programa para ler o nome e o número de matrícula de um aluno, como a seguir:

```
main ::  IO ()
main = do nome <- ask "Qual é o seu nome?  "
          matr <- ask "Qual é o seu número de matrícula?  "
          putStrLn ("Benvindo "++ nome ++ "!")
          putStrLn ("Seu número de matrícula é "++ matr)
```

A função `getLine`, usada anteriormente, é também definida em termos de uma ação mais elementar `getChar ::  IO Char`, que lê um caractere do teclado:

```
getLine = do ch <- getchar
             if (c == '\n')
               then return []
               else do cs <- getLine
                       return (c:cs)
```

A função `return`, usada na última linha acima, encapsula um valor de tipo `a` em um valor do tipo `IO a`, ou seja[2], ou seja, tem tipo `return ::  a -> IO a`.

Exercícios:

1. Escreva um programa que lê uma linha, a partir do teclado, verifica se ela contém apenas caracteres alfabéticos e imprime essa linha na tela, com as palavras em ordem inversa. Caso a linha contenha algum caractere não alfabético, imprime uma mensagem de erro.

2. Escreva um programa que pergunta ao usuário o seu nome e telefone e imprime na tela a informação obtida, em uma única linha.

3. Escreva um programa que lê várias linhas a partir do teclado, e imprime cada linha lida, com os caracteres convertidos para maiúsculas, até que seja digitada uma linha nula.

1.4 Lendo e exibindo valores

As ações `getLine` e `putStr` podem ser usadas para ler e escrever strings de caracteres. Para valores de outros tipos, devemos usar as funções `readLn` e `print`. Por exemplo, podemos ler um número inteiro do seguinte modo:

```
leInt ::  IO(Int)
leInt = do putStr "Digite um valor inteiro:  "
           readLn
```

A função `leInt` poderia também ser escrita do seguinte modo:

[2]Ao contrário do que ocorre em Java, `return` de fato não retorna uma valor, mas apenas converte um valor de tipo `a` em um valor de tipo `IO a`.

```
leInt2 :: IO(Int)
leInt2 = do putStr "Digite um valor inteiro: "
            n <- getLine
            return (read n)
```

Com base nessas definições, poderíamos escrever o seguinte programa para ler dois números inteiros e imprimir as soma desses números:

```
main :: IO ()
main = do n1 <- leInt
          n2 <- leInt
          putStr "A soma é: "
          print (n1+n2)
```

As funções `read`, `readLn` e `print` possuem as seguintes assinaturas:

```
read   :: Read a => String -> a
readLn :: Read a => a
print  :: Show a -> a -> IO ()
```

Ainda outra maneira de definir a função `leInt` seria:

```
leInt3 :: IO(Int)
leInt3 = do putStr "Digite um valor inteiro: "
            readIO
```

A função `readIO :: Read a => IO a` é uma combinação de `getLine` e `read`, exceto que ela propaga uma *exceção*, caso o string lido por `getLine` não represente um valor numérico. Veremos como uma exceção pode ser tratada, na seção 1.6, a seguir.

Podemos também usar as funções `readLn` e `print` para ler e imprimir quaisquer outros valores cujos tipos são instâncias da classe `Show`. Por exemplo, o programa a seguir lê um valor de ponto flutuante `x` e imprime a lista de todos os valores de `0` a `x`, em intervalos de `0.1`:

```
main :: IO ()
main = do putStr "Digite um valor: "
          x <- readLn
          print [0,0.1..x]
```

Duas outras funções são definidas na linguagem para operações de leitura a partir do dispositivo de entrada padrão:

```
getContents :: IO String
interact :: (String -> String) -> IO ()
```

A função `getContents` retorna toda a entrada digitada pelo usuário como um único string, que é lido de forma *lazy*, à medida que requerido.

A função `interact` tem como argumento uma função do tipo `String -> String`. Toda a entrada lida do dispositivo pardrão é passada como argumento para essa função e o string resultante é impresso no dispositivo de saída padrão. Por exemplo, o programa

```
main = interact (filter isUpper)
```

imprime na tela apenas as letras maiúsculas do string lido do teclado.

Exercícios:

- Defina uma função `leIntList` que lê para uma seqüência de valores inteiros do dispositivo de entrada padrão, até que seja digitado o valor `0`, e retorna a lista dos valores lidos.
- Refaça o exercício anterior, supondo que os números devem ser digitados todos em uma única linha.
- Defina um programa que lê uma valor inteiro positivo n e imprime a lista de pares (i, i^2), para valores de i no intervalo $1 \leq i \leq n$.

1.5 Entrada e Saída em Arquivos

Haskell faz interface com o mundo externo por meio de um *sistema de arquivos* abstrato – uma coleção de *arquivos* que podem ser organizados em *diretórios*. Nomes de arquivos e diretórios são objetos do tipo `String`, e podem especificar o *path* completo até o arquivo, ou apenas o nome do arquivo relativo ao diretório corrente. O formato do nome do arquivo corresponde ao utilizado no sistema operacional Unix.

Arquivos podem ser abertos para leitura, escrita ou leitura/escrita. Essa operação associa ao arquivo um *handler* (do tipo `Handle`), que é usado para posteriores referências ao arquivo em operações de leitura e escrita. Três *handlers* são automaticamente associados a arquivos no início da execução de um programa: `stdin` – dispositivo de entrada padrão, associado ao teclado; `stdout` – dispositivo de saída padrão, associado ao console; e `stderr` – dispositivo de erro padrão, também associado ao console.

Algumas operações básicas de leitura e escrita em carquivos, definidas no módulo `IO`, são apresentadas a seguir:

```
type File = String

writeFile  ::  File -> String -> IO ()
appendFile ::  File -> String -> IO ()
readFile   ::  File -> IO String
```

A função `writeFile` cria um novo arquivo, com o nome especificado pelo primeiro argumento, e escreve nesse arquivo o string passado como segundo argumento. Caso o arquivo especificado já exista, seu conteúdo é reescrito. A função `appendFile`, ao invés de reescrever o conteúdo do arquivo, simplesmente grava no final do mesmo o string passado como argumento. Finalmente, a função `readFile` lê o conteúdo do arquivo, retornando-o como um string.

O programa a seguir efetua a cópia do conteúdo de um arquivo para outro arquivo, ilustrando o uso das funções acima:

```
module CopyArq where
  import IO

  main ::  IO ()
  main = do putStr "Digite o nome do arquivo de entrada:  "
            ifile <- getLine
            putStr "Digite o nome do arquivo de saída:  "
            ofile <- getLine
            s <- readFile ifile
            writeFile ofile s
```

As funções `readFile`, `writeFile` e `appendFile` são implementadas em termos de funções mais elementares definidas no módulo `IO`. Algumas das funções disponíveis nesse módulo são relacionadas a seguir. Consulte o manual da linguagem para uma descrição mais detalhada dessas funções.

```
openFile    ::  File -> IOMode -> IO Handle
hClose      ::  IO Handle -> IO ()

hFileSize   ::  Handle -> IO Integer

hisOpen     ::  Handle -> IO Bool
hisClosed   ::  Handle -> IO Bool
hisReadable ::  Handle -> IO Bool
hisWritable ::  Handle -> IO Bool
hisSeekable ::  Handle -> IO Bool
hisEOF      ::  Handle -> IO Bool

hSetBuffering:  Handle -> BufferMode -> IO ()
gGetBuffering:  Handle -> IO BufferMode
hFlush      ::  Handle -> IO ()
hGetPosn    ::  Handle -> IO HandlePosn
hSetPosn    ::  HandlePosn -> IO ()
hSeek       ::  Handle -> SeekMode -> Integer -> IO ()

hGetChar    ::  Handle -> IO Char
hGetLine    ::  Handle -> IO String
hGetContents::  Handle -> IO String
hPutChar    ::  Handle -> Char -> IO ()
hPutStr     ::  Handle -> String -> IO ()
hPutStrLn   ::  Handle -> String -> IO ()
hPrint      ::  Show a => Handle -> a -> IO ()
```

1.5.1 Exemplo

O exemplo a seguir ilustra a implementação de um programa que lê de um arquivo uma lista de compras de supermercado e imprime a nota de compras correspondente, com o preço de cada produto e o valor total da compra. Consideramos os seguintes tipos:

```
type Cents       = Int
type PriceList   =[(String,Cents)]
type ShoppingList=[(String,Cents)]
```

Supomos que a lista de preços dos produtos do supermercado está armazenada em um arquivo, cujo conteúdo tem o formato exemplificado a seguir:

```
[ ("Café",230),
  ("Arroz",576),
  ("Suco",320),
  ⋮
  ("Pão de Forma",120)]
```

A seguinte função obtém o conteúdo deste arquivo:

```
readPriceList ::  File -> IO PriceList
readPriceList fname = do contents <- readFile fname
                         return (read contents)
```

A função que define a interação do programa com o usuário consiste em um *loop*, que aguarda a entrada de um item e a quantidade comprada do mesmo, terminando a interação quando for digitado um string nulo:

```
readShoppinList ::  IO ShoppingList
readShoppinList = do putStr "Digite um item de compra:  "
                     item <- getLine
                     if item ==
                       then return []
                       else do putStr "Quantidade = "
                               q <- readLn
                               items <- readShoppinList
                               return ((item,q):items)
```

Usando essas duas funções, o programa principal pode ser implementado do seguinte modo:

```
main ::  IO ()
main = do prices <- readPriceList
          shlist <- readShoppingList
          writeBillList "NotaDeCompra" prices shList
          appendFile "NotaDeCompra" ("Total = " ++ show (costs prices shlist))
```

Deixamos como exercício que você defina as funções:

- `costs :: PriceList ShoppinList -> Cents`, que retorna o valor total da nota de compra, dada a lista a lista de preços e a lista de compra;
- `writeBillList :: File -> PriceList -> ShoppingLIst -> IO ()`, que grava a nota de compra no arquivo passado como primeiro argumento, em um formato em que cada linha contém o nome de um item, a quantidade comprada e preço total correspondente.

1.6 Tratamento de Exceção

O sistema de entrada e saída em Haskell inclui um mecanismo simples de tratamento de exceções. Exceções provocadas por operações de entrada e saída são representadas por valores do tipo abstrato `IOError`. O tipo de erro ocorrido em uma operação de E/S pode ser testado por funções correspondentes, definidas no módulo `IO`:

```
isAlreadyExistsError ::  IOError -> Bool
isDoesNotExistError  ::  IOError -> Bool
isAlreadyInUseError  ::  IOError -> Bool
isFullError          ::  IOError -> Bool
isEOFError           ::  IOError -> Bool
isIllegalOperation   ::  IOError -> Bool
isPermissionError    ::  IOError -> Bool
isUserError          ::  IOError -> Bool
```

Exceções são criadas e capturadas por meio das seguintes funções:

```
ioError ::  IOError -> IO a
catch   ::  IO a -> (IOError -> IO a) -> IO a
```

A função `ioError` ocasiona uma exceção. A função `catch` estabelece um tratador que trata qualquer exceção ocorrida na porção de código protegida pelo `catch`. O tratador não é seletivo: ele captura todas as exceções ocorridas. A propagação de uma exceção deve ser feita explicitamente em um tratador, ocasionando-se novamente a exceção que não se desejar tratar. Por exemplo, na definição:

```
f g = catch g (\e -> if IO.isEOFError e
                       then return []
                       else ioError e)
```

a função `f` retorna `[]`, caso ocorra uma exceção de fim de arquivo durante a execução da função `g`, protegida pelo `catch`, ou propaga a exceção `e` para um tratador mais externo, caso ela seja algum outro tipo de exceção (diferente de fim de arquivo).

Quando uma exceção é propagada para fora do programa principal, o sistema de execução de Haskell imprime o erro correspondente e termina a execução do programa.

Alguns exemplos a seguir ilustram o tratamento de exceção. Como dissemos anteriormente, a função `readIO :: Read a => String -> IO a` retorna o valor de tipo `a` correspondente ao string passado como primeiro argumento, ocasionando uma exceção (de usuário), caso o string

não esteja no formato requerido para valores do tipo `a` em questão. Ilustramos, a seguir, como a função `leInt`, definida na seção 1.4, poderia ser modificada de modo a garantir a leitura de uma valor inteiro válido:

```
leInt ::  IO Int
leInt = do putStr "Digite um número inteiro:  "
           s <- getLine
           catch (readIO s) trataErro
        where trataErro e = if IO.isUserError e
                            then do putStrLn "Número inválido"
                                    leInt
                            else ioError e
```

O programa de cópia de arquivo apresentado na seção 1.5 poderia ser modificado do seguinte modo, de maneira a alertar o usuário caso o arquivo especificado como entrada seja inexistente:

```
module CopyArq where
import IO

main ::  IO ()
main = do putStr "Digite o nome do arquivo de entrada:  "
          ifile <- getLine
          putStr "Digite o nome do arquivo de saída:  "
          ofile <- getLine
          catch (copia infile ofile) trataErro
        where copy fi fo = do s <- readFile fi
                              writeFile fo s
              trataErro e = if IO.isDoesNotExistError e
                            then do putStrLn "Arquivo de entrada inexistente"
                                    main
                            else ioError e
```

Bibliografia

[1] Peyton-Jones, Simon, *Haskell 98 Language and Libraries: Revised Report*, January 2003, available at www.haskell.org.

[2] Thompson, Simom, *Haskell: The craft of functional programming*, second edition, Addison-Wesley, 2003.