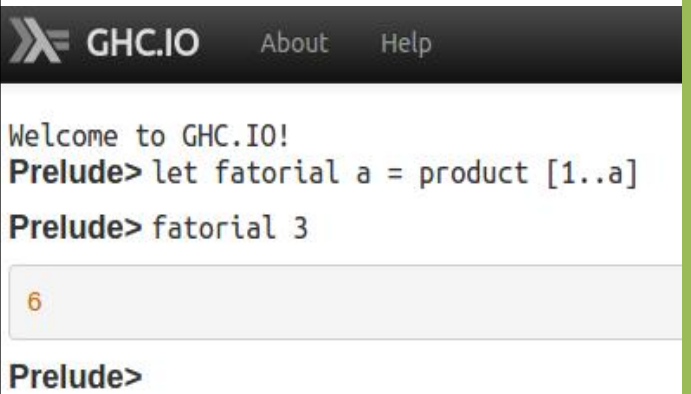

sumList [2,3,4,5]
= 2 + sumList [3,4,5]
= 2 + (3 + sumList [4,5])
= 2 + (3 + (4 + sumList [5]))
= 2 + (3 + (4 + (5 + sumList [])))
= 2 + (3 + (4 + (5 + 0)))
= 14

Programação Funcional

Manipulação de Arquivos

Matheus Carvalho Viana
matheuscviana@ufsj.edu.br

Dcomp
Departamento de Ciência da Computação

UFSJ
Universidade Federal de São João del-Rei

Introdução

Os programas manipulam dados para transformá-los em informações.

Esses dados precisam estar na memória RAM para serem manipulados. Por exemplo, quando o usuário digita um número para uma função calcular o seu fatorial.

Apesar de rápida, a memória RAM perde suas informações quando o computador é desligado.

Mesmo mantendo o computador ligado, devido ao tamanho da memória RAM, seu conteúdo é volátil, ou seja, está em constante modificação, a medida que os programas são processados e terminam sua execução.

Introdução

Portanto, é necessária uma forma de **persistir** os dados em arquivos armazenados em um dispositivo com maior quantidade de memória e capaz de manter o conteúdo mesmo estar ligado.

Em Haskell, a manipulação de arquivos é realizada por meio de comandos de entrada e saída pertencentes a biblioteca **`System.IO`**.

Lembre-se de importar essa biblioteca.

Arquivos de Texto ou Binários

Os programas podem trabalhar com **arquivos de texto** ou com **arquivos binários**.

Arquivo texto

- Os bits representam caracteres
- Podem ser lidos por editores de texto
- São "legíveis" para os humanos

Arquivos binários

- Os bits representam dados
- Utilizam qualquer sequência de bytes
- Processamento mais eficiente

Abertura e Fechamento de Arquivos

Para lidar com um arquivo, primeiramente é preciso abri-lo para obter o seu **descritor** (**`Handle`**), que é uma estrutura que mantém o arquivo armazenado na memória RAM.

Para abrir um arquivo de texto, usa-se o comando **`openFile`**, que recebe uma String que representa o caminho do arquivo e o modo de abertura e retorna o descritor do arquivo.

```
openFile :: FilePath -> IOMode -> IO Handle
```

Para abrir um arquivo binário, é necessário utilizar o comando **`openBinaryFile`**, cuja assinatura é semelhante a de **`openFile`**.

Abertura e Fechamento de Arquivos

Uma vez obtido o descritor do arquivo, podem ser feitas operações de leitura ou escrita sobre ele.

No final, é preciso fechar o arquivo com o comando **`hClose`** para que os dados não sejam perdidos quando o programa terminar sua execução.

```
hClose :: Handle -> IO ()
```

Esse comando recebe o descritor e retorna vazio.

Modos de Abertura

Na abertura do arquivo, também é necessário definir de que modo isso será feito. O modo de abertura é definido pelo tipo **`IOMode`**.

Valor IOMode	Permite	Posição Inicial	Detalhes
`ReadMode`	Leitura	Início do arquivo	O arquivo deve existir.
`WriteMode`	Escrita	Início do arquivo	Cria o arquivo, caso ele não exista, ou sobrescreve seu conteúdo, caso já exista.
`ReadWriteMode`	Leitura e Escrita	Início do arquivo	Cria o arquivo no caso dele não existir, ou mantém seu conteúdo, caso já exista.
`AppendMode`	Escrita	Fim do arquivo	Cria o arquivo no caso dele não existir, ou mantém seu conteúdo, caso já exista.

Leitura de Arquivos

Os principais comandos para ler o conteúdo de um arquivo são:

Assinatura do Comando	Descrição
`hGetChar :: Handle -> IO Char`	Lê um caractere do arquivo.
`hGetLine :: Handle -> IO String`	Lê uma linha do arquivo.
`hGetContents :: Handle -> IO String`	Lê todo o conteúdo do arquivo*.

Obs.: hGetContents retorna uma lista de caracteres usando a avaliação preguiçosa, em que cada caractere do arquivo é lido sob demanda.

Leitura de Arquivos

Por exemplo, o programa abaixo abre o arquivo exemplo.txt somente para leitura, lê e imprime o seu conteúdo e, no final, fecha o arquivo.

```
import System.IO

main = do
    handle <- openFile "exemplo.txt" ReadMode
    contents <- hGetContents handle
    putStr contents
    hClose handle
```

Escrita em Arquivos

Os principais comandos para escrever conteúdo em um arquivo são:

Assinatura do Comando	Descrição
`hPutChar :: Handle -> Char -> IO ()`	Escreve um caractere no arquivo.
`hPutStr :: Handle -> String -> IO ()`	Escreve uma String no arquivo.
`hPutStrLn :: Handle -> String -> IO ()`	Escreve uma String no arquivo.
`hPrint :: Show a => Handle -> a -> IO ()`	Escreve um valor de qualquer tipo pertencente a classe Show no arquivo.

Obs.: **`hPutStrLn`** e **`hPrint`** adicionam o caractere '\n' no final da String passada como entrada.

Escrita em Arquivos

Por exemplo, o programa abaixo abre o arquivo exemplo.txt somente para escrita, solicita que o usuário digite o seu conteúdo, escreve no arquivo e, no final, fecha o arquivo.

```
import System.IO

main = do
    handle <- openFile "exemplo.txt" WriteMode
    putStrLn "Digite o conteúdo do arquivo:"
    conteudo <- getLine
    hPutStrLn handle conteudo
    hClose handle
```

Manipulação de Arquivos

O descritor de um arquivo mantém um ponteiro que indica a posição atual de leitura/escrita desse arquivo.

Quando um arquivo é aberto, esse ponteiro tem o valor 0. Exceto no AppendMode, que abre o arquivo com o ponteiro posicionado no final.

Cada vez que uma leitura é feita, o sistema operacional retorna o valor do dado apontado por este ponteiro e incrementa o ponteiro para apontar para o próximo dado a ser lido.

O comando **`hTell`** pode ser utilizado para ver a posição corrente deste ponteiro em relação ao início do arquivo (posição zero).

```
hTell :: Handle -> IO Integer
```

Manipulação de Arquivos

O comando **hSeek** permite trocar o valor da posição atual.

```
hSeek :: Handle -> SeekMode -> Integer -> IO ()
```

O parâmetro **SeekMode** pode ter três valores:
AbsoluteSeek: o deslocamento do ponteiro ocorre a partir do início do arquivo;
RelativeSeek: o deslocamento do ponteiro ocorre a partir da posição atual;
SeekFromEnd o deslocamento do ponteiro ocorre a partir da posição final do arquivo.

O parâmetro Integer indica o tamanho do deslocamento.

Manipulação de Arquivos

O comando **`hIsEOF`** retorna **`True`** se o ponteiro do descritor esteja no fim do arquivo, ou **`False`**, caso não esteja.

Por exemplo, o programa a seguir lê um arquivo linha a linha e a escreve em outro arquivo com o conteúdo em maiúsculo.

```
import System.IO
import Data.Char (toUpper)

main :: IO ()
main = do
  entrada <- openFile "entrada.txt" ReadMode
  saida <- openFile "saida.txt" WriteMode
  loop_principal entrada saida
  hClose entrada
  hClose saida
```

Manipulação de Arquivos

```
loop_principal :: Handle -> Handle -> IO ()
loop_principal entrada saida = do
  fim_de_arquivo <- hIsEOF entrada
  if fim_de_arquivo
  then return ()
  else do
    inpStr <- hGetLine entrada
    hPutStrLn saida (map toUpper inpStr)
    loop_principal entrada saida
```

Exercícios

1. Crie uma função que solicita do usuário o caminho de um arquivo e retorna o tamanho dele em caracteres.
2. Crie uma função que solicita do usuário o caminho de um arquivo e retorna o tamanho dele em linhas.
3. Crie uma função que solicita do usuário o caminho de um arquivo e permite que ele escreva mais linhas ao final do arquivo. Para finalizar, o usuário deve digitar “EOF”.
4. Crie uma função que solicita que o usuário digite “sum” para realizar uma soma ou “sub” para realizar uma subtração. Em seguida, ele solicita que o usuário digite dois operandos e realiza a operação selecionada, imprimindo o resultado na tela no formato “a op b = s”, onde op pode ser ‘+’ ou ‘-’, a e b são os operandos e r é o resultado. O usuário pode repetir quantas vezes quiser até digitar “fim” no lugar da operação. O programa deve escrever em um arquivo todas as operações realizadas no formato “a op b = s”, cada uma em uma linha.