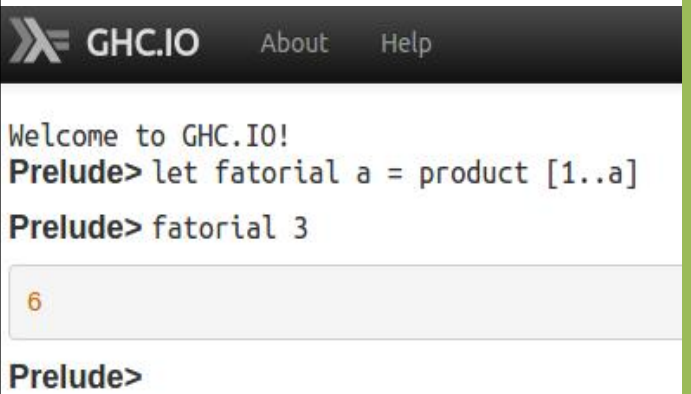

sumList [2,3,4,5]
= 2 + sumList [3,4,5]
= 2 + (3 + sumList [4,5])
= 2 + (3 + (4 + sumList [5]))
= 2 + (3 + (4 + (5 + sumList [])))
= 2 + (3 + (4 + (5 + 0)))
= 14

Programação Funcional

Entrada e Saída

Matheus Carvalho Viana
matheuscviana@ufsj.edu.br

Dcomp
Departamento de Ciência da Computação
UFSJ
Universidade Federal de São João del-Rei

Introdução

Haskell é conhecida por ser uma linguagem puramente funcional.

Na programação funcional, uma função não pode alterar algum estado, como mudar o conteúdo de uma variável.

A única coisa que uma função pode fazer é retornar algum resultado com base nos parâmetros de entrada.

Embora isso possa parecer um pouco limitante, na verdade é mais seguro, pois isso impede a existência de efeitos colaterais nas linguagens funcionais.

Introdução

O fato das funções não serem capazes de mudar o estado é bom porque ajuda o programador a raciocinar sobre cada função de forma independente.

Porém, há situações em que uma função não realiza o cálculo de uma expressão e necessita alterar o estado de algo externo a ela.

Por exemplo, no caso de uma operação de I/O, que valor deve ser retornado?

Este retorno é necessário para que o paradigma seja respeitado. Caso contrário, ele é corrompido.

Introdução

Para resolver esse problema, Haskell possui um sistema muito inteligente para lidar com funções com efeitos colaterais, capaz de separar a parte do programa que é pura da parte que faz todo o trabalho sujo como falar com o teclado e a tela.

A solução adotada foi introduzir um tipo especial chamado **`Ação (Action)`**.

Quando o sistema detecta uma operação deste tipo, ele sabe que uma ação deve ser executada.

Existem ações primitivas, por exemplo escrever um caractere em um arquivo ou receber um caractere do teclado, mas também ações compostas como imprimir várias strings.

Introdução

Toda função em Haskell que realiza uma ação é chamada de **comando**.

As funções, cujos retornos sejam ações, também são chamadas de comandos.

Os comandos realizam ações e retornam um valor de um determinado tipo, que pode ser usado futuramente pelo programa.

Introdução

Em Haskell, toda função que recebe, opera e/ou retorna uma ação é chamada de **comando**.

Haskell provê o tipo polimórfico **`IO a`** para permitir que um programa faça alguma operação de **`I/O`** e retorne um valor do tipo **`a`**.

Haskell também provê um tipo **`IO ()`**, cujo único valor é **`()`** (siginifica vazio, como o tipo void do C). Uma função que retorna o tipo **`IO ()`** executa uma ação e retorna o valor **`()`**.

Existem muitas funções pré-definidas em Haskell para realizar ações, além de um mecanismo para sequencializá-las, permitindo que alguma ação do modelo imperativo seja realizada sem ferir o modelo funcional.

Hello, World!

Até agora, as funções criadas foram interpretadas no GHCi como se fossem estruturas individuais que não compõem um programa maior.

Para obter um programa compilado, é necessário que haja uma função main, cujo retorno é do tipo **`IO ()`**. Por exemplo:

```
main = putStrLn "Hello, World!"
```

Agora, no terminal digite o comando abaixo para compilar o arquivo.

ghc --make helloworld.hs

Funções de Entrada

A operação de leitura de um caractere (**Char**), a partir do dispositivo padrão de entrada, é descrita em Haskell pela função **getChar**:

```
getChar :: IO Char
```

Para ler uma string, a partir do dispositivo padrão de entrada, usa-se a função pré-definida **getLine**:

```
getLine :: IO String
```

Para ler uma valor de um tipo **a** qualquer, a partir do dispositivo padrão de entrada, usa-se a função pré-definida **readLn**:

```
readLn :: Read a => IO a
```

Funções de Saída

A função é **putStr** é a operação padrão em Haskell para impressão de um texto. Ela recebe uma string, escreve-a no dispositivo padrão de saída e retorna um valor do tipo **()**.

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

Há também a função **putStrLn**, cujo efeito é adicionar o caractere de nova linha ao fim da entrada passada para **putStr**:

```
putStrLn :: String -> IO ()
putStrLn = putStr . (++ "\n")
```

.

Funções de Saída

Repare que **`putStr`** e **`putStrLn`** aceitam somente uma **`String`** como entrada. Por isso, muitas vezes é necessário converter um valor para **`String`** usando a função **`show`**.

```
show :: Show a => a -> String
```

Outra opção é a função print, que faz a conversão e chama putStrLn.

```
print :: Show a => a -> IO ()
print = putStrLn . show
```

Funções de Saída

Tanto **`print`** quanto **`putStrLn`** saltam uma linha no final da impressão (isso não é visível quando são usadas de forma isolada no GHCi).

Mas existe um diferença entre **`print`** e **`putStrLn`**.

Quando recebe como entrada uma String, **`print`** mantém as aspas.

Já a função **`putStrLn`** não.

```
Prelude> print "abc"
"abc"
Prelude> putStrLn "abc"
abc
```

O Comando do

O comando **do** é um mecanismo flexível, construído para realizar duas operações sobre ações em Haskell:

1. a execução em sequência de ações de I/O;
2. a captura de valores retornados por ações de I/O, para repassá-los futuramente para outras ações do programa.

Por exemplo, a função **imprime** realiza duas ações: a primeira é escrever a string no dispositivo de saída padrão e a segunda é fazer com que o *prompt* salte para a próxima linha.

```
imprime :: String -> IO ()
imprime str = do putStr str
                 putStr "\n"
```

O Comando do

Quando se trata de ações IO, é normal que os programas solicitem entradas ao usuário e faça uso dessas entradas. Isto é feito por meio da nomeação dos resultados das ações.

Por exemplo, a função **`soma2num`** solicita que o usuário digite um número, e armazena a **`String`** digitada na variável **`x`**. Solicita outro número e armazena a **`String`** em **`y`**. Por fim, imprime a soma.

```
soma2num :: IO ()
soma2num =
  do putStr "Digite um número: "
     x <- getLine
     putStr "Digite outro número: "
     y <- getLine
     putStrLn ("Soma: " ++ show (read x + read y))
```

O Comando return

O comando return interrompe a sequência de ações, forçando a saída da função e fazendo com que ela retorne o valor passado ao return;

No slide seguinte, a função **`leiaAte`** lê sequencialmente o nome de cada aluno e suas 3 notas e mostre o nome do aluno e sua média aritmética. Se o usuário digitar **`"Final"`**, a função **`leiaAte`** é encerrada e retorna **`()`**.

A função **`getDouble`** solicita que o usuário digite um número como **`String`** e retorna essa **`String`** convertida para o tipo **`Double`**.

O Comando return

```
leiaAte :: IO ( )
leiaAte = do
  nome <- getLine
  if nome == "fim"
  then return ()
  else do n1 <- getDouble
          n2 <- getDouble
          n3 <- getDouble
          putStr (nome ++ show ((n1+n2+n3)/3.0) ++ "\n")
          leiaAte

getDouble :: IO Double
getDouble = do
  dado <- getLine
  return (read dado :: Double)
```

O Comando when

Esse comando recebe um valor booleano e uma ação de IO.

- Se o valor booleano for **`True`**, when retornará a ação de IO;
- Se for **`False`**, when retorna **`()`**.

Dentro de um bloco **`do`**, **`when`** parece uma declaração de fluxo de controle, mas na verdade é uma função normal.

```
import Control.Monad
nomes :: IO ()
nomes = do
    putStr "Digite um nome ou fim: "
    s <- getLine
    when (s /= "fim") $ do
        putStrLn ("Você digitou " ++ s)
        nomes
```

Para usar o comando when é necessário importar a biblioteca **`Control.Monad`**.

O Comando sequence

O comando **`sequence`** recebe uma lista de ações I/O, as executa na sequência e retorna uma ação I/O sobre uma lista contendo os resultados de cada ação IO recebida. Sua assinatura é:

```
sequence :: [IO a] -> IO [a]
```

O exemplo abaixo solicita três strings ao usuário e as imprime na sequência.

```
sequencia :: IO ()
sequencia = do
    rs <- sequence [getLine, getLine, getLine]
    print rs
```

O Comando forever

O comando **`forever`** recebe uma ação I/O e retorna outra ação I/O que repete a ação original para sempre.

Este commando pertence à biblioteca **`Control.Monad`**.

O programa abaixo fica solicitando sem parar que o usuário digite algo e imprime convertido para maiúsculo:

```
import Control.Monad  (forever)
import Data.Char  (toUpper)

main = forever $ do
    putStr "Digite algo: "
    l <- getLine
    putStrLn $ map toUpper l
```

Pressione Ctrl+C para interromper a execução ou você ficará preso a ela para sempre.

Exercícios

1. Explique o que faz o código **`sequence (map print [1,2,3,4,5])`**?

2. Altere a função leiaAte (slide 15), acrescentando informações de saída para tornar a função mais legível ao usuário. Inclua uma função main para chamar leiaAte e compile o programa.

3. Crie uma função que solicita do usuário uma string, imprime ela invertida e informa se a mesma é um palíndromo ou não.

4. Crie uma função que solicita do usuário uma sequência de 3 inteiros e imprime se cada um é par ou ímpar.

5. Faça um programa compilado que mostre ao usuário um menu com as opções sair do programa, somar 2 números e multiplicar 2 números e fatorial de um inteiro. O programa só encerra se o usuário optar por sair.