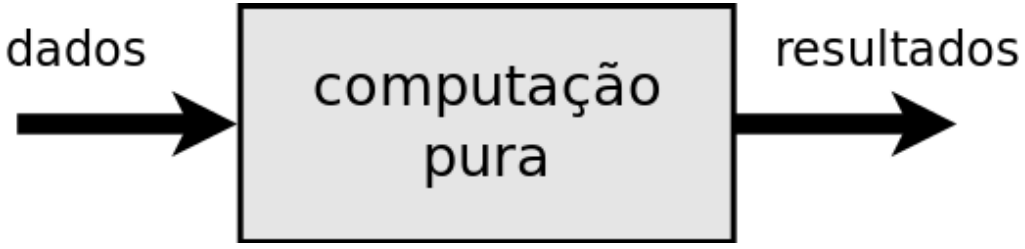
Programação Funcional
7ª Aula — Programas interativos

PROF. BONFIM AMARO JUNIOR

bonfimamaro@ufc.br

Motivação

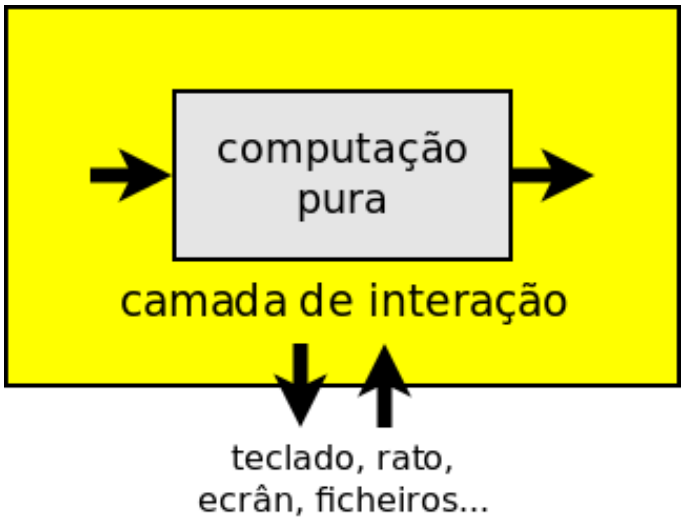
Até agora apenas escrevemos programas que efetuam computação pura, i.e., transformações funcionais entre valores.



Motivação (cont.)

Vamos agora ver como escrever programas interativos:

- lêm informação do teclado, ficheiros, etc.;
- escrevem no terminal ou em ficheiros;
- ...



Ações de I/O

Introduzimos um novo tipo `IO ()` para ações que, *se forem executadas*, fazem entrada/saída de dados.

Exemplos:

```
putChar 'A' :: IO ()                    -- imprime um 'A'
putChar 'B' :: IO ()                    -- imprime um 'B'

putChar :: Char -> IO ()                -- imprimir um carater
```

Encadear ações

Podemos combinar duas ações de I/O usando o operador de *sequênciação*:

```
(>>) :: IO () -> IO () -> IO ()
```

Exemplos:

```
(putChar 'A' >> putChar 'B') :: IO ()     -- imprimir "AB"
(putChar 'B' >> putChar 'A') :: IO ()     -- imprimir "BA"
```

Note que >> é associativo mas não é comutativo!

Encadear ações (cont.)

Em alternativa podemos usando a notação-*do*:

```
  putChar 'A' >> putChar 'B' >> putChar 'C'
=
  do {putChar 'A'; putChar 'B'; putChar 'C'}
```

Podemos omitir os sinais de pontuação usando a indentação:

```
do putChar 'A'
   putChar 'B'
   putChar 'C'
```

Execução

Para efetuar as ações de I/O definimos um valor `main` no módulo `Main`.

```
module Main where

main = do putChar 'A'
          putChar 'B'
```

Compilar e executar:

```
$ ghc Main.hs -o prog
$ ./prog
AB$
```

Execução (cont.)

Também podemos efetuar ações IO diretamente no `ghci`:

```
Prelude> putChar 'A' >> putChar 'B'
ABPrelude>
```

Definir novas ações

Vamos agora definir novas ações de I/O combinando ações mais simples.

Exemplo: definir `putStr` usando `putChar` recursivamente.

```
putStr :: String -> IO ()
putStr []     = ??
putStr (x:xs) = putChar x >> putStr xs
```

Como completar?

Ação vazia

```
putStr :: String -> IO ()
putStr []     = return ()
putStr (x:xs) = putChar x >> putStr xs
```

`return ()` é a ação vazia: *se for efetuada*, não faz nada.

IO

`IO` *a* é o tipo de ações que, *se forem executadas*, fazem entrada/saída de dados e devolvem um valor de tipo *a*.

Exemplos:

```
putChar 'A' :: IO ()        -- escrever um 'A'; resultado vazio

getChar :: IO Char          -- ler um caracter; resultado Char
```

Ações IO pré-definidas

```
getChar :: IO Char                    -- ler um caracter
getLine :: IO String                  -- ler uma linha
getContents :: IO String              -- ler toda a entrada padrão

putChar :: Char -> IO ()              -- escrever um carater
putStr :: String -> IO ()             -- escrever uma linha de texto
putStrLn :: String -> IO ()           -- idem com mudança de linha

print :: Show a => a -> IO ()         -- imprimir um valor

return :: a -> IO a                   -- ação vazia
```

Combinando leitura e escrita

Usamos <- para obter valores retornados por uma ação I/O.

Exemplo: ler e imprimir caracteres até obter um fim-de-linha.

```
main :: IO ()
main = do x<-getChar
          putChar x
          if x=='\n' then return () else main
```

Combinando leitura e escrita (cont.)

Outro exemplo:

```
boasvindas :: IO ()
boasvindas = do putStr "Como te chamas? "
                nome <- getLine
                putStr ("Bem-vindo, " ++ nome ++ "!\n")
```

Valores de retorno

Podemos usar `return` para definir valores de retorno de ações.

```
boasvindas :: IO String
boasvindas
   = do putStr "Como te chamas? "
        nome <- getLine
        putStr ("Bem-vindo, " ++ nome ++ "!\n")
        return nome
```

Valores de retorno (cont.)

Outro exemplo: definir `getLine` usando `getChar`.

```
getLine :: IO String
getLine = do x<-getChar
             if x==’\n’ then
                return []
              else
                do xs<-getLine
                   return (x:xs)
```

Jogo *Hi-Lo*

Exemplo maior: um jogo de perguntas-respostas.

- o computador escolhe um número secreto entre 1 e 100;
- o jogador vai fazer tentativas de advinhar;
- para cada tentativa o computador diz se é *alto* ou *baixo*;
- a pontuação final é o número de tentativas.

Jogo *Hi-Lo* (cont.)

```
Tentativa? 50
Demasiado alto!
Tentativa? 25
Demasiado baixo!
Tentativa? 35
Demasiado alto!
Tentativa? 30
Demasiado baixo!
Tentativa? 32
Acertou em 5 tentativas.
```

Jogo *Hi-Lo* (cont.)

Vamos decompor em duas partes:

- **main** escolhe o número secreto e inicia o jogo;
- **jogo** função recursiva que efetua a sequência perguntas-respostas.

Programa

```
module Main where

import Data.Char(isDigit)
import System.Random(randomRIO)

main = do x <- randomRIO (1,100)  -- escolher número aletório
          n <- jogo 1 x           -- começar o jogo
          putStrLn ("Acertou em " ++ show n ++
                    " tentativas")
```

Programa (cont.)

```
jogo :: Int -> Int -> IO Int
jogo n x                          -- n: tentativas, x: número secreto
  = do { putStr "Tentativa? "
       ; str <- getLine
       ; if all isDigit str then
           let y = read str in
           if y>x then
             do putStrLn "Demasiado alto!"; jogo (n+1) x
           else if y<x then
             do putStrLn "Demasiado baixo!"; jogo (n+1) x
                else return n
         else do putStrLn "Tentativa inválida!"; jogo n x
       }
```

Ações são valores

As ações IO são valores de primeira classe:

- podem ser argumentos ou resultados de funções;
- podem passados em listas ou tuplos;
- ...

Isto permite muita flexibilidade ao combinar ações.

Ações são valores (cont.)

Exemplo: uma função para efetuar uma lista de ações por ordem.

```
seqn :: [IO a] -> IO ()
seqn []     = return ()
seqn (m:ms) = m >> seqn ms
```

Ações são valores (cont.)

Exemplos de uso:

```
> seqn [putStrLn s | s<-["ola", "mundo"]]
ola
mundo

> seqn [print i | i<-[1..5]]
1
2
3
4
5
```

Sumário

- Programas reais necessitam de combinar interação e computação pura
- Em Haskell fica explícito nos tipos quais as funções que fazem interação e quais são puras.
- A notação-`do` e o tipo `IO` é usada para:
  - ler e escrever no terminal e em ficheiros;
  - estabelecer comunicações de rede;
  - serviços do sistema operativo
    (ex: obter data e hora do relógio de sistema);
  - bibliotecas escritas em outras linguagens (ex: C/C++).

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 os caracteres em ordem inversa. Caso a linha contenha algum caractere não alfabético, imprime uma mensagem de erro.

Dica: A biblioteca import Data.Char possui a função "isLetter" (Retorna verdade ou falso para um caractere ser letra)

Exemplo:
*Main> main
*Main> Digite uma palavra :
*Main> AsFs
Saída : sFsA

*Main> main
*Main> Digite uma palavra :
*Main> AsFs12
Saída : ERRO

Exercícios

2) 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.

Exemplo:
*Main> main
*Main> Digite uma sentença
*Main> programacao
*Main> PROGRAMACAO
*Main> Digite uma sentença
*Main>

Dica: A biblioteca import Data.Char possui a função “toUpper” (Converte os caracteres minúsculos para o correspondente maiúsculos)

Exercícios

3) Defina uma função leIntList que lê para uma sequência de valores inteiros do dispositivo de entrada padrão, até que seja digitado o valor 0, retorna a lista dos valores lidos e outra função SomaValoresInt que soma de todos os valores.
Função para ler Inteiros

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

4) Defina um programa que lê um valor inteiro positivo n e imprime a lista de pares (i, i*i), para valores de i no intervalo [1,n]

Exemplo:

```
*Main> main
*Main> Digite um valor inteiro:
*Main> 5
*Main> ( 1 , 1 ) , ( 2 , 4 ) , ( 3 , 9 ) , ( 4 , 16 ) , ( 5 , 25)
```