## Interativos

Programas

### Programas interativos

#### Programas interativos podem:

- exibir mensagens para o usuário
- obter valores informados pelo usuário

De forma geral um programa poderá trocar informações com o restante do sistema computacional para

- obter dados do sistema computacional
- gravar dados no sistema computacional

Em linguagens imperativas as operações de entrada e saída produzem **efeitos colaterais**, refletidos na atualização de variáveis globais que representam o estado do sistema de computação.

#### Exemplo de programa interativo em C

Programa que obtém dois caracteres digitados pelo usuário e exibe-os em letras maiúsculas na tela:

```
#include <stdio.h>
#include <ctype.h>

Int main(vold)
{
    char x=getchar();
    char y=getchar();
    printf("%c%c\n", toupper(x), toupper(y));
    return 0;
}
```

Supondo que o usuário informe os caracteres 'A' e 'b', a execução do programa produzirá a seguinte interação:

```
Ab
AB
```

# Exemplo de programa interativo em C(cont.)

- A aplicação de função getchar() retornavalores diferentes mesmo quando chamada com os mesmos argumentos (nenhum argumento, neste caso). A primeira chamada retorna 'A' e a segunda chamada retorna 'h'.
- Isto acontece porque getchar() utiliza uma variável global representando o dispositivo de entrada padrão (stdin). Durante a chamada da função esta variável é atualizada (efeito colateral), removendo o próximo caractere disponível na entrada e retornando-o como resultado.

# Exemplo de programa interativo em C(cont.)

 Assim, quando a função getchar() é chamada novamente, o próximo caractere disponível na entrada padrão é o segundo caractere digitado pelo usuário. Esta informação está na variável global que representa o dispositivo de entrada padrão (stdin).

#### Linguagens puras

Em **linguagens puras** o valor retornado por uma função depende única e exclusivamente dos seus argumentos.

Portanto toda vez que uma função é aplicada a um dado argumento, o resultado é o mesmo.

Assim não é possível implementar uma função que lê um caracter da mesma maneira que em linguagens impuras, como C.

#### Exemplo:

```
let x = getchar()
    y = getchar()
in ...
```

Em uma linguagem pura os valores de x e y serão iguais, uma vez que são definidos aplicando a função getchar ao mesmo argumento.

#### O mundo

- Para interagir com o usuário, precisamos de uma representação do sistema de computação onde o programa está sendo executado: o mundo (world).
- O mundo é formado por todas as informações no contexto de execução da aplicação, incluindo:
  - dispositivo de entrada padrão (o teclado)
  - dispositivo de saída padrão (a tela)
  - sistema de arquivos (arquivos em disco)
  - conexões de rede
  - gerador de números pseudo-aleatórios (usa uma semente que depende do sistema, como por exemplo o horário atual)

#### Modificando o mundo

Em **linguagens impuras** o mundo (ou parte dele) corresponde a uma variável global atualizável.

Uma função impura que interage com o mundo pode alterar esta variável, de forma que uma aplicação posterior da função ao mesmo argumento pode retornar um valor diferente.

Em uma linguagem pura não há a possibilidade de alterar uma variável.

Uma função pura que interage com o mundo tem um argumento e um resultado adicionais que representam o mundo antes e o mundo depois da interação.

### Ações de entrada e saída

- Uma ação de entrada e saída (E/S) é uma atividade que representa uma interação com o mundo.
- Uma ação de E/S pode ser executada para interagir com o mundo e retornar um valor obtido através desta interação.
  - Em Haskell O a é o tipo das ações de entrada e saída que interagem com o mundo e retornam um valor do tipo a.
- IO a é um tipo abstrato, logo sua representação não está disponível nos programas. Haskell provê:
  - algumas ações de entrada e saída primitivas, e
  - um mecanismo para combinar ações de entrada e saída.

### A função putchar

#### putChar :: Char -> IO()

putChar é uma função que recebe um caracter e resulta em uma ação de E/S que, quando executada, interage com o mundo inserindo o caractere na saída padrão e retorna a tupla vazia ().

Quando executada, a ação putchar x apenas insere x na saída padrão e não há nenhum valor interessante para ser retornado. Como toda ação deve retornar um valor quando executada, a tupla vazia () é usada.

Exemplo: O valor da expressão

#### putChar'H'

é uma ação de E/S que, quando executada, interage com o mundo inserindo o caracter 'H' na saída padrão e retorna ().

## A função putstr

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

A função putstr recebe uma string e resulta em uma ação de E/S que, quando executada, interage com o mundo inserindo a string na saída padrão e retorna a tupla vazia.

## A função putstrLn

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

A função putstrln recebe uma string e resulta em uma ação de E/S que, quando executada, interage com o mundo inserindo a string seguida do caracter '\n' na saída padrão e retorna a tupla vazia.

### A função print

```
print :: Show a => a -> 10 ()
```

A função print recebe um valor e resulta em uma ação de E/S que, quando executada, insere na saída padrão o valor convertido para string, seguido de mudança de linha, e retorna a tupla vazia.

A conversão para string é feita usando a função

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

Portanto o tipo do valor deve ser instância da classe Show.

## A ação getChar

#### getChar :: 10 Char

A ação de E/S **getchar**, quando executada, interage com o mundo extraindo o próximo caracter disponível da entrada padrão e retorna este caracter.

A ação getChar levanta uma exceção (que pode ser identificada pelo predicado isEOFError do módulo IO) se for encontrado fim de arquivo.

## A ação getLine

#### getLine :: 10 String

A ação de E/S getLine, quando executada, interage com o mundo extraindo a próxima linha disponível na entrada padrão e retorna esta linha.

A ação getLine pode falhar com uma exceção se encontrar o fim de arquivo ao ler o primeiro caracter.

### A ação getContents

#### getContents :: IO String

A ação de E/S getContents , quando executada, interage com o mundo extraindo todos os caracteres da entrada padrão e retorna a string formada pelos caracteres.

## A ação readLn

#### readLn **:: Read** a ⇒ IO a

- A ação de E/S readLn, quando executada, interage com o mundo extraindo a próxima linha disponível na entrada padrão e retorna um valor obtido dessa string.
- A conversão da string para o valor é feita usando uma função similar à função read, com a diferença de que se a conversão falhar o programa não termina, mas uma exceção é levantada no sistema de E/S.
- Portanto o tipo do valor deve ser instância da classe Read.

#### Programa em Haskell

- Quando uma ação de E/S é executada?
- Um programa em Haskell é uma ação de E/S.
- Executar o programa implica em executar a ação de E/S que o constitui. Um programa é organizado como uma coleção de módulos.
- Um dos módulos deve ser chamado Main e deve exportar a variável main, do tipo IO t, para algum t.
- Quando o programa é executado, a ação main é executada, e o seu resultado (do tipo t) é descartado.

## Exemplo de programa em Haskell

#### Exibir um caracter.

```
module Main (main) where

main :: IO ()
main = putChar'A'
```

#### Quando o programa é executado:

- main recebe (automaticamente) como argumento o mundo existente antes de sua execução,
- 2.realiza ações de entrada e saída
- 3.resultando em uma tupla vazia (nenhum valor interessante é produzido), e
- 4.produzindo um novo mundo que reflete o efeito das ações de entrada e saída realizadas.

## Preparando e executando um programa em Haskell

- Grave o código fonte do programa em um arquivo texto, digamos putchar-a.hs
- Compile o programa(por exemplo usando o Glasgow Haskell Compiler em um terminal):

```
$ ghc --make putchar-a
[1 of 1] Compiling Main ( putchar-a.hs, putchar-a.o )
Linking putchar-a ...
```

3. Execute o programa já compilado:

```
$ ./putchar-a
A
```

### Sequenciamento de ações

- Sendo IO a um tipo abstrato, como poderíamos combinar duas ações em sequência?
- Exemplo: como exibir os caracteres 'A' e 'B' em sequência?
- Haskell tem uma forma de expressão (expressão do) que permite combinar ações de entrada e saída a serem executadas em sequência.
- Exemplo:

```
do { putChar'A'; putChar'B'}
```

### Expressão do

Uma **expressão do** permite combinar vários ações de E/S de forma **sequencial**. Uma **expressão do** é da forma

```
do { comando_1; ...; comando_n; expressao }
```

com n ≥ 0.

expressão é uma ação de E/S. Cada comando; pode ser da forma:

- expressao
   uma ação de E/S cujo retorno é ignorado
- padrao <- expressao</p>
  uma ação de E/S cujo retorno é casado com o padrão indicado. O escopo das variáveis introduzidas no casamento de padrão estende-se até o final da expressão do. Se o casamento falhar, toda a ação falha.
- let declarações permite fazer declarações cujo escopo se estende até o final da expressão do. É semelhante à expressão

**let** declaracoes **in** expressao porém sem a expressão.

### Expressão do (cont.)

O valor da **expressão do** é uma ação de E/S formada pela combinação sequencial das ações de E/S que a compõem.

Quando a **expressão do** é **executada**, as ações que a compõem são executadas em sequência, e o valor retornado pela expressão do é o valor retornado pela última ação.

## Exemplo de expressão do

Exibe três caracteres na saída padrão.

```
module Main (main) where

main :: 10()
main = do { putChar'F'; putChar'i'; putChar'm'}
```

# Exemplo de expressão do(cont.)

Exibe três caracteres na saída padrão.

```
module Main (main) where

main :: IO()
main = do { putChar'F';
    putChar'i';
    putChar'm'
}
```

# Regra de *layout* com a expressão do

- A expressão do pode usar a regra de layout da mesma maneira que let, where e case.
- Assim as chaves { e } e os pontos-e-vírgula ; podem ser omitidos, sendo substituídos por uso de indentação adequada.
- Neste caso, cada comando que compõe a expressão do deve começar na mesma coluna e, se continuar em linhas subsequentes, deve sempre ocupar as colunas à direita da coluna onde iniciou.

# Regra de *layout* com a expressão do(cont.)

#### Exemplo:

Exibe três caracteres na saída padrão.

```
module Main (main) where

main :: IO()
main = do putChar'F'
    putChar'i'
    putChar'm'
```

## Exemplo: ler um caracter

Obter um caracter da entrada padrão.

```
module Main (main) where

main :: IO Char

main = getChar
```

## Exemplo: ler e exibir um caracter

Obter um caracter da entrada padrão e exibi-lo na saída padrão.

# Exemplo: ler e exibir um caracter (v2)

Ler um caracter e exibi-lo em minúsculo e em maiúsculo.

### Exemplo: saudação

Ler o nome do usuário e exibir uma saudação.

#### Exercícios

#### Exercício

Escreva um programa em Haskell que solicita ao usuário para digitar uma frase, lê a frase (uma linha) da entrada padrão e testa se a string lida é uma palíndrome, exibindo uma mensagem apropriada.

## Exemplo: soma de dois números

Ler dois números e exibir a soma dos mesmos.

```
module Main (main) where

main :: 10()
main = do putStrLn"Digite um número:"
    s1 <- getLine
    putStrLn"Digite outro número:"
    s2 <- getLine
    putStr"Soma dos números digitados:"
    putStrLn (show (read s1+read s2))</pre>
```

# Exemplo: soma de dois números (v2)

Ler dois números e exibir a soma dos mesmos.

#### Exercícios

#### Exercício

Escreva um programa que solicita ao usuário três números em ponto flutuante, lê os números, e calcula e exibe o produto dos números.

#### Exemplo: Soma de dois números

Ler dois números e exibir a soma dos mesmos.

Execução do programa onde o usuário informa os números 34 e 17:

```
34
17
Digite um número: Digite outro número: Soma dos números digitados: 51
```

O que aconteceu de errado?

## Saída buferizada

- A saída para o dispositivo padrão de saída é bufferizada: o sistema operacional mantém uma área da memória (buffer) onde armazena os caracteres a serem enviados para o dispositivo de saída.
- Geralmente os caracteres enviados para a saída padrão somente são transferidos para o dispositivo de saída quando o **buffer** estiver cheio.
- Este mecanismo reduz o número de acesso aos dispositivos de saída (que são muito mais lentos que o processador), melhorando o desempenho da aplicação.
- Por este motivo as mensagens não aparecem imediatamente quando o programa anterior é executado.

# Esvaziamento do *buffer* de saída

A função hFlush (definida no módulo **System.IO**) recebe um manipulador de arquivo (*handle*) e resulta em uma ação de E/S que, quando executada, faz com que os itens armazenados no *buffer* de saída do manipulador sejam enviados imediatamente para a saída.

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

O tipo **Handle** (definido no módulo **System**. **IO**) é um tipo abstrato que representa um dispositivo de E/S internamente para o Haskell.

O módulo **System**. **IO** define variáveis que representam alguns dispositivos padrões:

```
stdin :: Handle -- entrada padrão
stdout :: Handle -- saída padrão
stderr :: Handle -- saída de erro padrão
```

## Exemplo: Soma de dois

Ler dois números e exibir a soma dos mesmos.

```
module Main (main) where
import System.IO (stdout, hFlush)
main :: 10()
main = do putStr"Digite um número:"
          hFlush stdout
          s1 <- getLine
          putStr"Digite outro número:"
          hFlush stdout
          s2 <- getLine
          putStr"Soma dos números digitados:"
          putStrLn (show (read s1+read s2))
```

Execução do programa onde o usuário informa os números 34 e 17:

```
Digite um número: 34
Digite outro número: 17
Soma dos números digitados: 51
```

# Modos de *bufferização*

A função hsetBuffering (definida no módulo **System.10**) pode serutilizada para configurar o modo de *buferização* de um dispositivo.

```
hSetBuffering :: Handle -> BufferMode -> IO()
```

- O tipo BufferMode (definido no módulo System.IO) representa um modo de bufferização:
  - sem buferização:

#### **NoBuffering**

buferização por linha:

### LineBuffering

buferização por bloco:

### **BlockBuffering**

 Normalmente a saída para o dispositivo padrão é feita com buferização por linha.

# Modos de *bufferização* (cont.)

### A expressão

### hSetBuffering hdl mode

é uma ação que, quando executada, configura o modo de *bufferização* para o *handler* hd1.

Então podemos corrigir o problema no exemplo dado anteriormente adicionando a ação

```
hSetBuffering stdout NoBuffering
```

no começo da sequência de ações.

## Exemplo: Soma de dois números

Ler dois números e exibir a soma dos mesmos.

```
module Main (main) where
import System.IO (stdout, hSetBuffering, BufferMode(NoBuffering))
main :: 10()
main = do hSetBuffering stdout NoBuffering
          putStr"Digite um número:"
          s1 <- getLine
          putStr"Digite outro número:"
          s2 <- getLine
          putStr"Soma dos números digitados:"
          putStrLn (show (read s1+read s2))
```

Execução do programa onde o usuário informa os números 34 e 17:

```
Digite um número: 34
Digite outro número: 17
Soma dos números digitados: 51
```

Escreva um programa em Haskell que solicita ao usuário uma temperatura na escala Fahrenheit, lê esta temperatura, converte-a para a escala Celsius, e exibe o resultado. Para fazer a conversão, defina uma função celsius :: **Double->Double** que

recebe a temperatura na escala Fahrenheit e resulta na temperatura correspondente na escala Celsius

Use a seguinte equação para a conversão:

$$C = \frac{5}{9} \times (F - 32)$$

onde F é a temperatura na escala Fahrenheit e C é a temperatura na escala Celsius. Use a função celsius na definição de main.

A digitação da temperatura em Fahrenheit deve ser feita na mesma linha onde é exibida a mensagem que a solicita.

## Exemplo: peso ideal

Escrever um programa em Haskell que recebe a altura e o sexo de uma pessoa e calcula e mostra o seu peso ideal, utilizando as fórmulas constantes na tabela a seguir.

sexo	peso ideal
masculino	72.7 × h – 58
feminino	62.1 × h - 44.7

onde h é a altura da pessoa.

## Exemplo: peso ideal(cont.)

```
module Main (main) where
Import System.IO (stdout.hSetBuffering.BufferMode(NoBuffering))
import Data.Char (toUpper)
main :: 10()
main = do hSetBuffering stdout NoBuffering
          putStr"Altura:"
          h <- readin
          putStr"Sexo (f/m):"
          s <- aetLine
          case toUpper (head s) of
            'F' -> putStrLn ("Peso ideal:"++show (62.1*h-44.7))
            'M' -> putStrLn ("Peso ideal:"++show (72.7*h-58))
                putStrLn"Sexo inválido"
```

## Exemplo: média de 3 notas

Faça um programa que receba três notas de um aluno, e calcule e mostre a média aritmética das notas e a situação do aluno, dada pela tabela a seguir.

média das notas	situação
menor que 3	reprovado
entre 3 (inclusive) e 7	exame especial
acima de 7 (inclusive)	aprovado

## Exemplo: média de 3

```
module Main (main) where
import System.IO (stdout, hSetBuffering, BufferMode(NoBuffering))
prompt :: Read a =>String->IO a
prompt msa = do putStr msa
                read! n
main :: 10()
main = do hSetBuffering stdout NoBuffering
          n1 <- prompt "Nota 1:"
          n2 <- prompt "Nota 2:"
          n3 <- prompt "Nota 3:"
          Let media = (n1+n2+n3)/3
          putStrLn ("Média:"++show media)
          putStr "Situação:"
          if media<3</pre>
            then putStrLn "reprovado"
            elseif media<7
                   then putStrLn "exame especial"
                   else putStrLn "aprovado"
```

# Exemplo: raízes da equação do segundo grau

Faça um programa que leia os coeficientes de uma equação do segundo grau e calcule e mostre suas raízes reais, caso existam.

# Exemplo: raízes da equação do segundo grau(cont.)

```
module Main (main) where
import System.IO (stdout,hSetBuffering,BufferMode(NoBuffering))
Raizes2grau a b c
  |d>0 = [ (-b+sqrt d)/(2*a), (-b-sqrt d)/(2*a) ]
  |d=0| = [-b/(2*a)]
  otherwise = []
 where d = b \wedge 2 - 4 * a * c
prompt mensagem = do { putStr mensagem: readLn }
main = do hSetBuffering stdout NoBuffering
          putStrLn"Cálculo das raízes da equação do segundo grau"
          putStrLn"a x^2 + b x + c = 0"
          a <- prompt"Coeficiente a:"
          b <- prompt"Coeficiente b:"</pre>
          c <- prompt"Coeficiente c:"</pre>
          case raizes2 grau a b c of
            [r1.r2] >> putStrLn ("Raizes:"++show r1++"e"++show r2)
            [r] -> putStrLn ("Raíz:"++show r)
                       putStrLn"Não há raízes reais"
            []->
```

A prefeitura de Contagem abriu uma linha de crédito para os funcionários estatutários.

O valor máximo da prestação não poderá ultrapassar 30% do salário bruto. Fazer um programa que permita entrar com o salário bruto e o valor da prestação, e informar se o empréstimo pode ou não ser concedido.

### Exercício

Crie um programa que leia a idade de uma pessoa e informe a sua classe eleitoral:

não eleitor: abaixo de 16 anos; eleitor obrigatório: entre 18 (inclusive) e 65 anos;

eleitor facultativo: de 16 até 18 anos e acima de 65 anos (inclusive).

Faça um programa que apresente o menu a seguir, permita ao usuário escolher a opção desejada, receba os dados necessários para executar a operação, e mostre o resultado.

Opções:

Imposto
 Novo salário

Classificação

Digite a opção desejada:

Verifique a possibilidade de opção inválida. Na **opção 1** receba o salário de um funcionário, calcule e mostre o valor do imposto sobre o salário usando as regras a seguir:

## Exercícios(cont.)

salário	taxa de imposto
Abaixo de R\$500,00	5%
De R\$500,00 a R\$850,00	15%
Acima de R\$850,00	10%

Na **opção 2** receba o salário de um funcionário, calcule e mostre o valor do novo salário, usando as regras a seguir:

salário	aumento
Acima de R\$1.500,00	R\$25,00
De R\$750,00 (inclusive) a R\$1.500,00 (inclusive)	R\$50,00
De R\$450,00 (inclusive) a R\$750,00	R\$75,00
Abaixo de R\$450,00	R\$100,00

Na **opção 3** receba o salário de um funcionário e mostre sua classificação usando a tabela a seguir:

salário	classificação
Até R\$750,00 (inclusive)	mal remunerado
Acima de R\$750,00	bem remunerado

## A função return

### return :: a -> 10 a

Às vezes é necessário escrever uma ação de E/S que não faz nenhuma interação com o mundo e retorna um valor previamante especificado.

A função return recebe um valor e resulta em uma ação de E/S que não interage com o mundo e retorna o valor.

# Exemplo: exibir uma sequência

Faça um programa que exiba todos os números naturais pares menores ou iguais a 30.

# Exemplo: exibir uma sequência(cont.)

```
module Main (main) where

mostraLista :: Show a => [a] -> 10 ()

mostraLista xs|null xs = return()

| otherwise = do putStrLn (show (head xs))

mostraLista (tail xs)

main :: 10 ()

main = mostraLista [0,2..30]
```

# Exemplo: somar uma sequência

Escreva um programa que repetidamente lê uma sequência de números (um por linha) até encontrar o valor zero, e mostra a soma dos números lidos.

```
-- solução sem recursividade de cauda
module Main (main) where
main = do putStrLn"Digite uma sequência de números (um por linha)"
          putStrLn"Para terminar digite o valor zero"
          soma <- lerESomar
          putStr"A soma dos números digitados é"
          putStrLn (show soma)
lerESomar = do n <- readLn
               if n==0
                 then return 0
                 else do somaResto <- lerESomar
                         return (n+somaResto)
```

```
-- solução com recursividade de cauda
module Main (main) where
main = do putStrLn"Digite uma sequência de números (um por linha)"
          putStrLn"Para terminar digite o valor zero"
          soma <- leresomar0
          putStr"A soma dos números digitados é"
          putStrLn (show soma)
lerESomartotal = do n <- readLn</pre>
                     if n==0
                        then return total
                       else lerESomar (total+n)
```

```
-- solução que separa a leitura do processamento
  sem recursividade de cauda
module Main (main) where
main = do putStrLn"Digite uma sequência de números (um por linha)"
          putStrLn"Para terminar digite o valor zero"
          lista <- lerrista
          putStr"A soma dos números digitados é"
          putStrLn (show (sum lista))
lerLista = do x <- readIn
              \mathbf{x} = 0
                then return []
                else do resto <- lerLista
                         return (x:resto)
```

```
-- solução que separa a leitura do processamento
-- com recursividade de cauda
module Main (main) where
main = do putStrLn"Digite uma sequência de números (um por linha)"
          putStrLn"Para terminar digite o valor zero"
          lista <- lerLista []
          putStr"A soma dos números digitados é"
          putStrLn (show (sum lista))
lerLista xs =do x <- readLn
                  if y==0
                   then return (reverse xs)
                   else lerLista (x:xs)
```

Faça um programa que leia um número natural *n*, e então leia outros *n* números e calcule e exiba a soma destes números.

### Exercício

Faça um programa que leia uma seqüência de números não negativos e determine a média aritmética destes números. A entrada dos números deve ser encerrada com um número inválido (negativo).

Um funcionário de uma empresa recebe aumento salarial anualmente. O primeiro aumento é de 1,5% sobre seu salário inicial. Os aumentos subsequentes sempre correspondem ao dobro do percentual de aumento do ano anterior. Faça uma aplicação onde o usuário deve informar o salário inicial do funcionário, o ano de contratação e o ano atual, e calcula e exibe o seu salário atual.

Faça uma aplicação para fechamento das notas de uma disciplina. Cada aluno recebe três notas nas atividades desenvolvidas. O usuário deverá informar a quantidade de alunos na turma, e em seguida as notas de cada aluno. Calcule e exiba:

- a média aritmética das três notas de cada aluno,
- a situação do aluno, dada pela tabela seguinte.

média aritmética	situação
até 3	reprovado
entre 3 (inclusive) e 7	exame especial
acima de 7 (inclusive)	aprovado

- a média da turma
- > percentual de alunos aprovados
- > percentual de alunos em exame especial
- > percentual de alunos reprovados

e mostre:

Faça um programa para corrigir provas de múltipla escolha que foram aplicadas em uma turma de alunos. O usuário deverá informar:

- o gabarito (as respostas corretas de cada questão) da prova
  - > a matrícula e as respostas de cada aluno da turma
- As notas devem ser normalizadas na faixa de zero a dez. Assim para calcular a nota obtida em uma prova, divida a soma dos pontos obtidos (um ponto para cada resposta correta) pelo número de guestões na prova, e multiplique o resulado por dez. Calcule
  - 1.a matrícula e a nota de cada aluno
  - 2.a taxa (em porcentagem) de aprovação, sabendo-se que a nota mínima para aprovação é sete.