Linguagens de Programação

Programação Funcional e Haskell
Programação Interativa

Thiago Alves

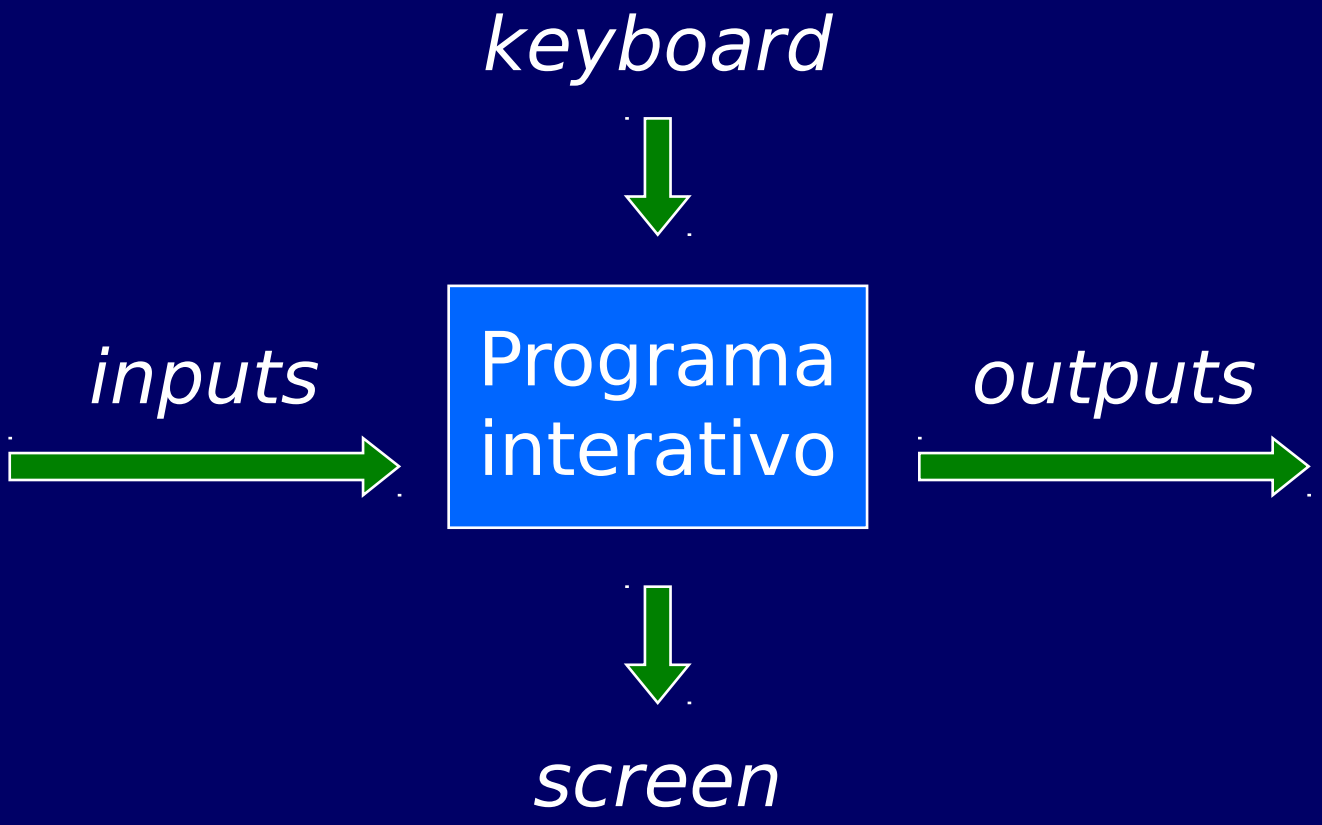
Introdução

Vimos como Haskell pode ser usado para escrever programas batch que recebem todas suas entradas no início e retornar todas as saídas no fina.

Introdução

Um programa batch não possui interação do usuário enquanto está rodando.

Queremos usar Haskell para escrever programas interativos que recebem dados do teclado e escrevem na tela enquanto estão rodando.



O Problema

Programas em Haskell são funções matemáticas puras:

- Com a mesma entrada possui a mesma saída

```
sumsqreven :: [Int] → Int
sumsqreven = sum . map (^2) . filter even
```

- Programas em Haskell não possuem efeitos colaterais.

O Problema

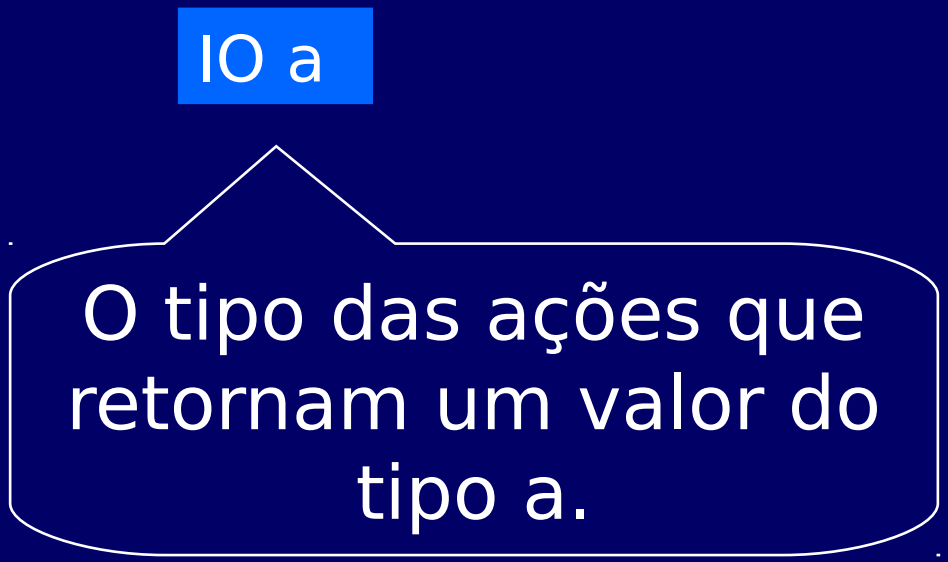
Ler dados do teclado e escrever na tela são efeitos colaterais:

- Programas interativos possuem efeitos colaterais:

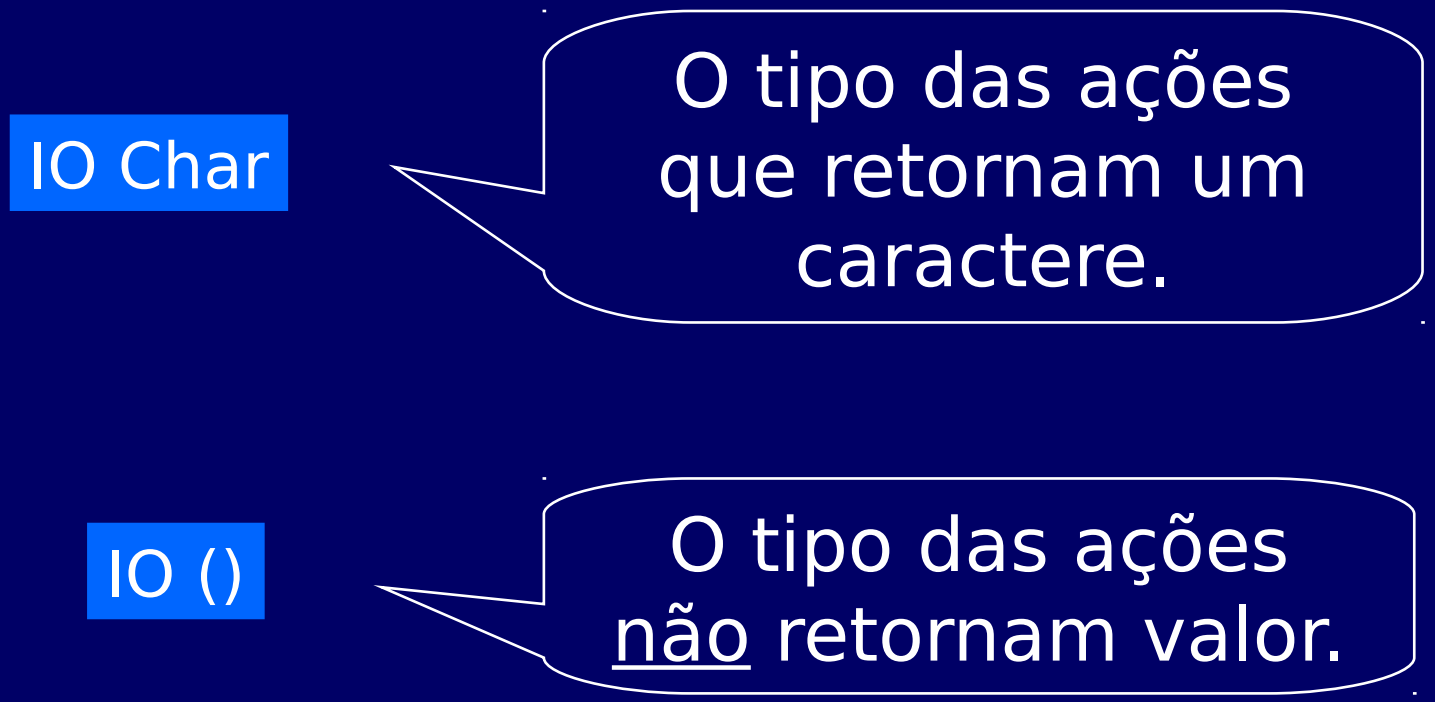
```
int fun(int a) {
  int b;
  scanf("%d",&b);
  return a + b;
}
```

A Solução

Programas interativos podem ser escritos em Haskell usando tipos para distinguir expressões puras de ações impuras que podem ter efeitos colaterais.



Exemplo:



- () é o tipo de tuplas sem componentes.

Ações Básicas

A biblioteca padrão fornece várias ações, incluindo as seguinte primitivas:

- A ação getChar recebe um caractere do teclado, mostra na tela e retorna o caractere como resultado:

```
getChar :: IO Char
```

- A ação putChar c escreve o caractere c na tela e não retorna valor:

```
putChar :: Char → IO ()
```

```
> putChar 'a'
a
```

```
main = putChar 'z'
```

Sequenciamento

Pode combinar ações de IO usando o operador de sequenciamento

```
(>>) :: IO a -> IO b -> IO b
```

em que x >> y é a ação que realiza x, desconsiderando o resultado e depois eealiza y e retorna seu resultado

```
putChar 'a' >> putChar 'B' >> putChar 'c'
```

Queremos encadear ações em que o resultado da primeira afete a segunda

```
(>>=) :: IO a -> (a → IO b) -> IO b
```

em que x >>= f é a ação que realiza x, passa o resultado para f e realiza uma segunda ação e retorna seu resultado

```
getChar >>= (\x -> putChar x)
```

```
main = getChar
       >>= \x -> putChar x
       >> putChar 'b'
       >> getChar
```

```
main = getChar
       >>= \x -> getChar
       >> getChar
       >>= \y -> putChar x
       >> putChar y
       >> getChar
```

Uma sequencia de ações podem ser combinadas como uma única ação composta usando do:

```
do x <- getChar; getChar; putChar x
```

```
main = do x ← getChar
          getChar
          y ← getChar
          getChar
          putChar x
          putChar y
```

Primitiva de Retorno

- A ação return v retorna o valor v sem realizar nenhuma interação:

```
return :: a → IO a
```

Primitivas Derivadas

- Receber uma string do teclado:

```
getLine :: IO String
```

Primitivas Derivadas

- Receber uma string do teclado:

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

- Escrever string na tela:

```
putStr :: String → IO ()
```

- Escrever string na tela:

```
putStr :: String → IO ()
putStr [] = return ()
putStr (x:xs) = do putChar x
                   putStr xs
```

- Escrever string e mover para uma nova linha:

```
putStrLn :: String → IO ()
```

- Escrever string e mover para uma nova linha:

```
putStrLn :: String → IO ()
putStrLn xs = do putStr xs
                 putChar '\n'
```

Exemplos

Vamos fazer um programa que recebe um nome e mostra uma saudação:

Exemplos

Vamos fazer um programa que recebe um nome e mostra uma saudação:

```
main = do putStrLn “Qual seu nome?”
          nome ← getLine
          putStrLn (“Bem-vindo, ” ++ nome)
```

Exemplos

Vamos fazer um programa que recebe dois números e mostra a soma:

use a função
readLn :: Read a => IO a
que recebe um valor do teclado e retorna esse valor e a função
show :: (Show a) => a -> String
que converte para string

Exemplos

Vamos fazer um programa que recebe dois números e mostra a soma:

```
main = do putStrLn “Digite o primeiro: ”
          x1 ← readLn
          putStrLn “Digite o segundo: ”
          x2 ← readLn
          putStrLn (show (x1 + x2))
```

Exemplos

Podemos definir uma ação que pede para uma string ser digitada e mostra seu tamanho:

```
strlen :: IO ()
```

Exemplos

Podemos definir uma ação que pede para uma string ser digitada e mostra seu tamanho:

```
strlen :: IO ()
strlen  = do putStr "Enter a string: "
             xs ← getLine
             putStr "The string has "
             putStr (show (length xs))
             putStrLn " characters"
```

Exemplos

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

Exemplos

```
readSum :: IO int
readSum = do n ← readLn
             if n == 0 then
               return 0
             else
               do sumAux ← readSum
                  return (n + sumAux)
```

Exemplos

```
main = do
        putStrLn “Digite uma sequencia de néméros:”
        putStrLn “Para terminar, digite 0”
        totalSum ← readSum
        putStr “A soma é: ”
        putStr show(totalSum)
```

Jogo da Forca

Considere a seguinte versão do jogo da forca:

- Um jogador digita uma palavra secretamente.
- O outro tenta deduzir a palavra, entrando com uma sequência de tentativas.
- Para cada tentativa, o computador indica que letras na palavra secreta aparecem na tentativa.
- O jogo termina quando a tentativa estiver correta.

```
hangman :: IO ()
hangman  =
   do putStrLn "Think of a word: "
      word ← sgetLine
      putStrLn "Try to guess it:"
      play word
```

A ação getCh recebe um único caractere do teclado sem mostrar na tela:

```
import System.IO


getCh :: IO Char
getCh  = do hSetEcho stdin False
            x ← getChar
            hSetEcho stdin True
            return x
```

A ação sgetLine recebe uma string do teclado e mostra como um dash:

```
sgetLine :: IO String
sgetLine  = do x ← getCh
               if x == '\n' then
                  do putChar x
                       return []
               else
                  do putChar '-'
                       xs ← sgetLine
                       return (x:xs)
```

A função play é o loop principal que pede e processa as tentativas até o jogo terminar.

```
play :: String → IO ()
play word =
   do putStr "? "
      guess ← getLine
      if guess == word then
         putStrLn "You got it!"
      else
         do putStrLn (match word guess)
               play word
```

A função match indica que caracteres em uma string ocorrem em uma outra string:

```
match :: String → String → String
match xs ys = [if elem x ys then x else '-' | x ← xs]
```

Exemplo:

```
> match "haskell" "pascal"

"-as--ll"
```