# Linguagens de Programação

Programação Funcional e Haskell
Introdução

Thiago Alves

# Funções

Mapeamento que recebe um ou mais argumentos e produz um único resultado

- Nome da função, nomes dos argumentos e especificação de como o resultado deve ser obtido:

```
double x = x + x
```

# Funções

O resultado é obtido substituindo os argumentos na especificação da função:

```
double (double 2)
=   { aplicando o double interno }
double (2 + 2)
=   { aplicando + }
double 4
=   { aplicando double }
4 + 4
=   { aplicando + }
8
```

# Programação Funcional

- Programação funcional é um estilo de programação no qual os métodos básicos de computação são aplicações de funções a argumentos;

# Exemplo

Somando os inteiros 1 até 10 em Java:

```
int n = 10;
int total = 0;
for (int i = 1; i ≤ n; i++)
   total = total + i;
```

O método de computação é a mudança de valores armazenados usando atribuição de variáveis.

# Exemplo

Somando os inteiros 1 até 10 em Haskell:

```
soma 1 = 1
soma n = n + soma (n-1)

soma 10
```

O método de computação é a aplicação de funções.

# Exemplo

```
soma 3
=   { aplicando soma}
3 + soma (3-1)
=   { aplicando -}
3 + soma 2
=   { aplicando soma}
3 + 2 + soma (2-1)
=   { aplicando -}
3 + 2 + soma 1
=   { aplicando soma}
3 + 2 + 1
= 6
```

# Programação Funcional

- Em geral, funções são tipos primitivos:

```
twice f x = f (f x)
```

```
twice double 3
=   { aplicando twice}
double (double 3)
=   { aplicando double interno}
double 6
=   { aplicando double}
12
```

# Haskell

- GHC é a implementação principal do Haskell, e fornece um compilador e um interpretador;
- Existem outros compiladores como o hugs: www.haskell.org/hugs/

# Iniciando o GHCi

O interpretador pode ser iniciado do terminal digitando ghci:

```
$ ghci

GHCi, version X: http://www.haskell.org/ghc/  :? for help

Prelude>
```

Por exemplo, pode ser usado para realizar expressões numéricas simples:

```
> 2+3*4
14

> (2+3)*4
20

> sqrt (3^2 + 4^2)
5.0
```

# Funções da Biblioteca Padrão

Haskell vem com um grande número de funções da biblioteca padrão. Além das funções numéricas familiares como + e *, a bilioteca também fornece várias funções sobre listas.

- Selecionar o primeiro elemento de uma lista:

```
> head [1,2,3,4,5]
1
```

- Remover o primeiro elemento:

```
> tail [1,2,3,4,5]
[2,3,4,5]
```

- Selecionar o n-ésimo elemento:

```
> [1,2,3,4,5] !! 2
3
```

- Selecione os primeiros n elementos:

```
> take 3 [1,2,3,4,5]
[1,2,3]
```

- Remover os primeiros n elementos:

```
> drop 3 [1,2,3,4,5]
[4,5]
```

- Calcular o tamanho da lista:

```
> length [1,2,3,4,5]
5
```

- Reverter uma lista:

```
> reverse [1,2,3,4,5]
[5,4,3,2,1]
```

(1) Mostre como a função last que seleciona o último elemento da lista pode ser definida usando as funções vistas

(1) Mostre como a função last que seleciona o último elemento da lista pode ser definida usando as funções vistas

```
last xs = head (reverse [1,2,3,4,5])
```

(2) De forma similar, mostre como a função init que remove o último elemento da lista pode ser definida.

(2) De forma similar, mostre como a função init que remove o último elemento da lista pode ser definida.

```
init xs = reverse (tail (reverse [1,2,3,4,5]))
```

- Calcular o produto de uma lista de números:

```
> product [1,2,3,4,5]
120
```

- Concatenar duas listas:

```
> [1,2,3] ++ [4,5]
[1,2,3,4,5]
```

- Calcular a soma dos elementos de uma lista de números:

```
> sum [1,2,3,4,5]
15
```

# Aplicação de Funções

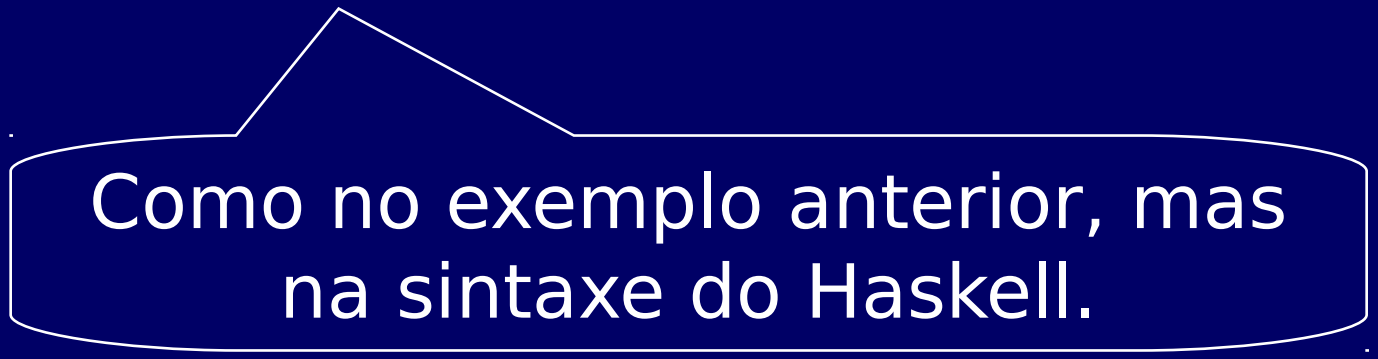
Na matemática, uma aplicação de função é denotada usando parênteses, e multiplicação é representada usando justaposição.

```
f(a,b) + c d
```

Aplique a função f em a e b, e adicione o resultado ao produto de c e d.

Em Haskell, aplicação de função é denotada usando espaço e multiplicação é representada usando *.

```
f a b + c*d
```



Além disso, aplicação de função tem maior prioridade que todos os outros operadores.

```
f a + b
```

Indica (f a) + b, em vez de
f (a + b).

# Exemplos

| Matemática | Haskell |
| --- | --- |
| f(x) | f x |
| f(x,y) | f x y |
| f(g(x)) | f (g x) |
| f(x,g(y)) | f x (g y) |
| f(x)g(y) | f x * g y |

# Scripts em Haskell

- Novas funções são definidas com um script, um arquivo de texto contendo uma sequência de definições;
- Por convenção, scripts em Haskell têm sufixo .hs.

# Primeiro Script

Inicie um editor, digite as seguintes funções, e salve o script como test.hs:

```
double x    = x + x

quadruple x = double (double x)
```

Inicie o GHCi com o script:

```
$ ghci test.hs
```

Agora a biblioteca padrão e o arquivo test.hs estão carregados. Funções de ambos podem ser usadas:

```
> quadruple 10
40

> take (double 2) [1,2,3,4,5,6]
[1,2,3,4]
```

No editor, adicione as seguintes definições:

```
factorial n = product [1..n]

average ns = sum ns `div` length ns
```

Note:

- div está no meio de crases;
- x `f` y é um açúcar sintático para f x y.

Use o comando reload para usar as novas definições:

```
> :reload
Reading file "test.hs"

> factorial 10
3628800

> average [1,2,3,4,5]
3
```

| Comando | Significado |
| --- | --- |
| :load *name* | load script *name* |
| :reload | reload current script |
| :type *expr* | show type of *expr* |
| :? | show all commands |
| :quit | quit GHCi |

# Requerimentos para Nomes

- Nomes de funções e argumentos precisam começar com letra minúscula:

myFun  fun1  arg_2  x’

- Por convenção, argumentos listas têm o sufixo s. Por exemplo:

xs  ns  nss

# A Regra de Layout

Em uma sequência de definições, cada definição deve começar precisamente na mesma coluna:

```
a = 10
b = 20
c = 30
```

```
a = 10
 b = 20
c = 30
```

```
 a = 10
b = 20
 c = 30
```

A regra de layout evita a necessidade de sintaxe explícita para indicar o agrupamento de definições.

```
a = b + c
   where
     b = 1
     c = 2
d = a * 2
```


```
a = b + c
    where
      {b = 1;
       c = 2}
 d = a * 2
```

Agrupamento implícito

Agrupamento explícito

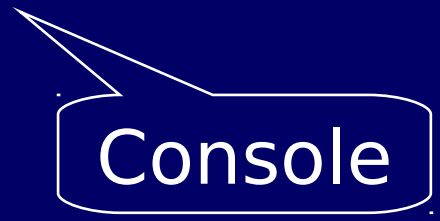
# Comentários

```
-- Fatorial de um inteiro positivo
factorial n = product [1..n]

{-
double comentada
double x = x + x
-}
```

# Entrada e Saída - Introdução

```
> putStrLn “Hello, Haskell”
“Hello, Haskell”
> print (5 + 4)
9
> print (1 < 2)
True
> do n <- readLn; print (2*n)
3
6
> do ling <- getLine; putStr ling
haskell
haskell
```

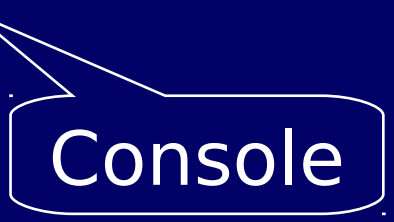


# Entrada e Saída - Introdução

```
factorial n = product [1..n]

main = do n <- readLn
          print (factorial n)
          name <- getLine
          putStr (“Hello, ” ++ name)
```


```
> main
5
120
haskell
Hello, haskell
```



# Entrada e Saída - Introdução

```
factorial n = product [1..n]

main = do n <- readLn
          print (factorial n)
```

Script

```
$runhugs  test.hs
4
24
```

hugs

```
$runhaskell  test.hs
5
120
```

GHCi