Programação Funcional

(COMP0393)

Leila M. A. Silva

dcomp
departamento de computação

Tipos Básicos e Definições

(COMP0393)

Aula 3

dcomp
departamento de computação

Programação Funcional

- Baseada em expressões e funções (*expression oriented*);
- Funções são definidas como equações;
- Avaliação das funções é realizada da mesma forma que na matemática;
- Provê elementos para se programar em vários níveis de granularidade:
  - Funções de alta ordem
  - Polimorfismo paramétrico
  - Avaliação preguiçosa
  - Tipos recursivos (algébricos)
  - Tipos Abstratos de Dados (TADs)
  - Type Classes
  - Módulos
- Raciocínio formal com os programas
- Prototipação

dcomp departamento de computação

Programação Funcional

- Por que usar uma linguagem do paradigma funcional?
  - Código menor, mais claro e mais fácil de manter
  - Poucos erros, alta confiabilidade.
  - Menor “gap” semântico entre os programadores e a linguagem.
  - Tempo menor de produção.

dcomp
departamento de computação

Haskell

- Criada por Peyton Jones e Hughes em 1998
- Haskell B. Curry – pioneiro do lambda cálculo (teoria matemática de funções)

Definições

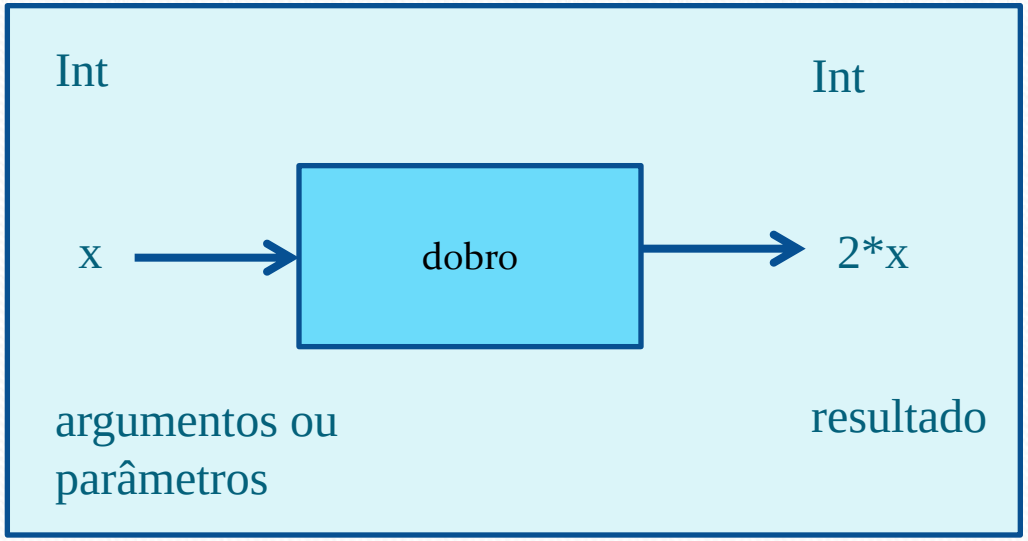
- Um programa em Haskell consiste de um conjunto de **definições.**
- Uma definição associa um **nome** (ou **identificador**) a um valor de um **tipo** particular.
- Definições simples seriam:

```
nome :: tipo
nome = expressão

idade :: Int
idade = 2019-1993
```

Definições

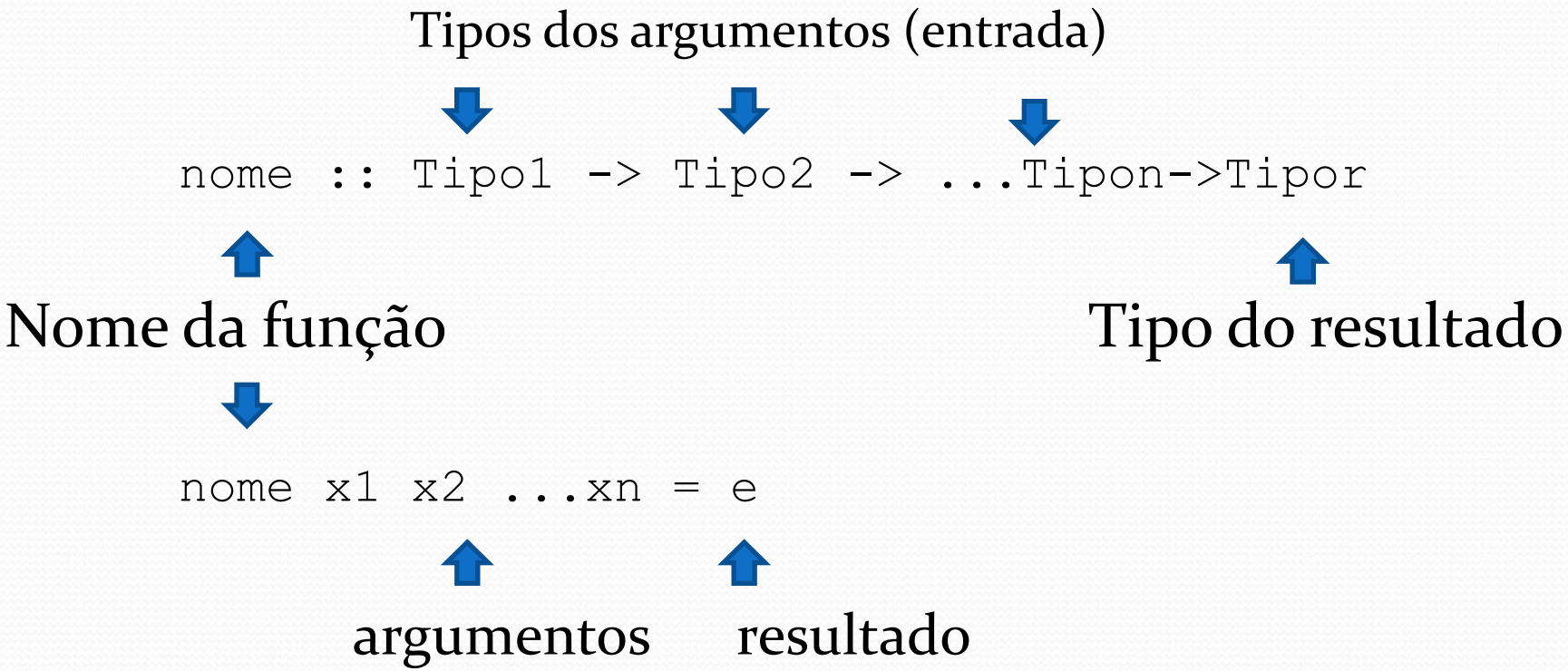
- Definições de funções também consideram os tipos de entrada e saída da função

```
dobro :: Int -> Int
dobro x  = 2*x

dobro 3 = 2*3
dobro (2+3) = 2*(2+3)
```



Definições

- Declaração e definição de funções de forma geral



Tipos Básicos

- O que é um tipo?
  - Uma coleção de valores aos quais podemos aplicar funções/operações sobre estes valores
- Por que linguagens modernas usam tipos?
  - Programas são mais confiáveis (seguros) pois erros de programação podem ser descobertos mais facilmente
  - Melhoram a legibilidade
    - Informação relevante para documentação
  - Programas mais eficientes

dcomp departamento de computação

Tipos Básicos

- Em Haskell toda expressão é de algum tipo
  - Restringe a uso coerente
  - Permite checagem estática de tipos => detecção precoce de erros
  - Tipos podem ser explicitados pelo programador
  - Sistema de inferência de tipos

Tipos Básicos

- Primitivos (ou atômicos)
  - Valores atômicos
  - inteiros, caracteres, reais, booleanos, enumerados, ...
- Tipos compostos (ou estruturados)
  - Arrays, listas, ...
- OBS: Nomes de tipos em Haskell iniciam sempre em maiúsculo e nomes de funções em minúsculo.

dcomp departamento de computação

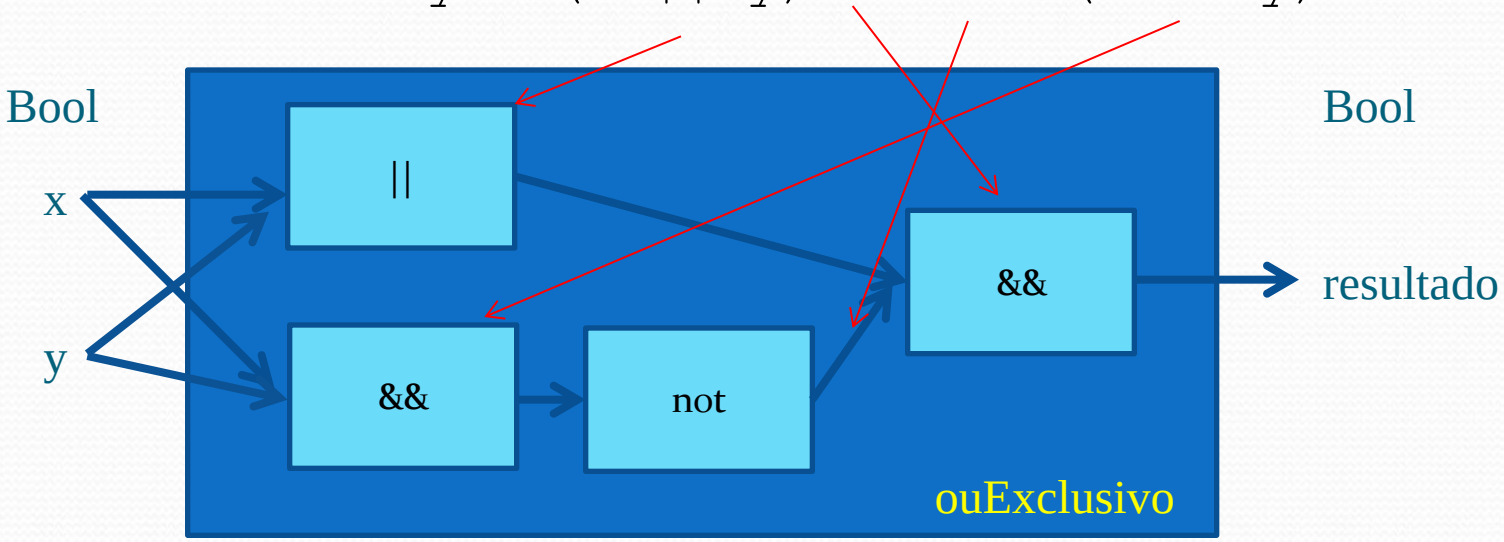
Tipos Primitivos

- Bool
  - Valores: True, False
  - Operações: && (e), || (ou), not (negação), == (igualdade), /= (desigualdade)
  - Tipos Booleanos podem ser usados como argumentos ou resultados de funções. Também são o resultado de algumas expressões relacionais que iremos ver adiante.

Tipos Primitivos

- Bool
  - Ex:

```
ouExclusivo :: Bool -> Bool -> Bool
ouExclusivo x y = (x || y) && not (x && y)
```



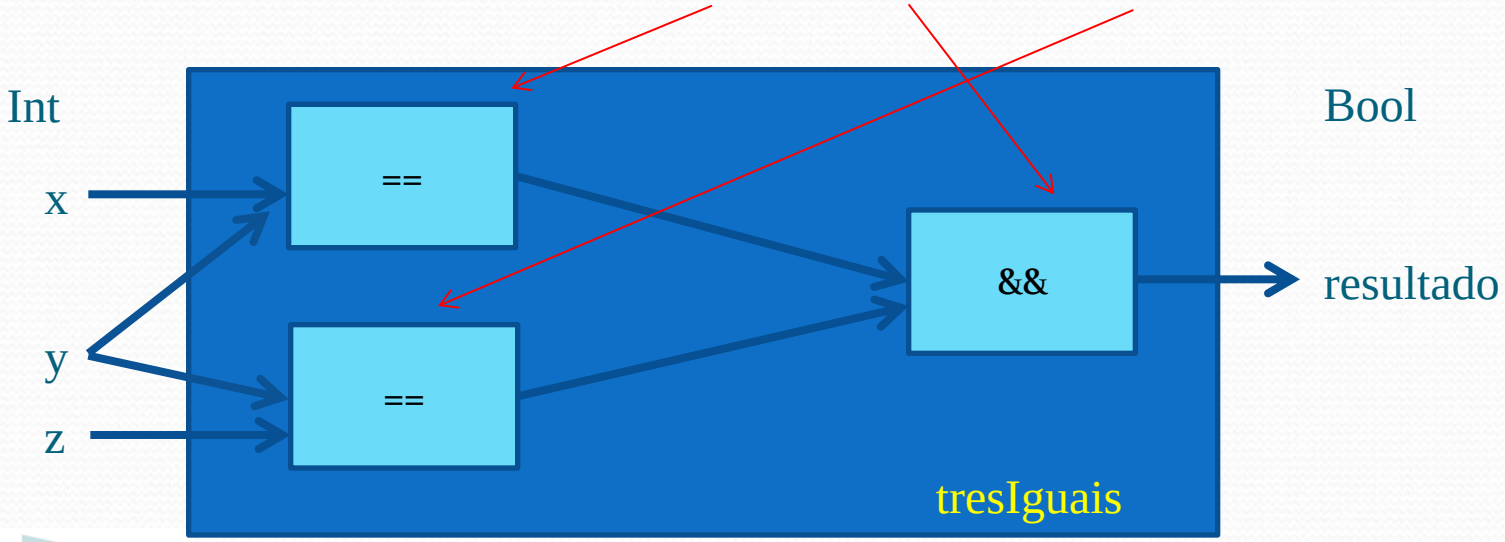
Tipos Primitivos

- Int (64 bits)
  - Literais: 0, 4, -345, 2147483647, maxBound::Int, ...
  - Operações: +, *, ^, -, <, <=, ==, /=, >=, >
  - Funções: div, mod, abs, negate, show, read
- Integer (ilimitado)

Tipos Primitivos

- Int
  - Ex:

```
tresIguais :: Int -> Int -> Int -> Bool
tresIguais x y z = (x == y) && (y == z)
```



Tipos Primitivos

- Sobrecarga de operadores (overloading)
  - Algumas operações são permitidas para tipos diferentes utilizando o mesmo símbolo. Por exemplo, ==
  - Para comparação de inteiros

```
(==) :: Int -> Int -> Bool
```

  - Para comparação de Booleanos

```
(==) :: Bool -> Bool -> Bool
```

Tipos Primitivos

- Char
  - 'a', 'b', ‘A‘,'\t', '\n', '\\', '\”, ‘\”’
  - Código para codificar caracteres como inteiros – ASCII
  - Ex: ‘A’ a ‘Z’ : códigos 65 a 90; ‘a’ a ‘z’: códigos 97 a 122
  - Operações: <, >, ==, \=, <=, >=
  - Funções de conversão: ord, chr

```
ord :: Char -> Int
chr :: Int -> Char

ehMinusculo::Char->Bool
ehMinusculo c = ('a' <= c) && (c <= 'z')
```

Tipos Primitivos

- Float e Double
  - 0.31416, -23.12, 2.45e+2
  - Operações: +, -, *, /, ^, **, <, <=, ...
  - Funções: abs, acos, asin, atan, cos, sin, tan, exp, log, ...

```
mediaTres :: Float -> Float -> Float -> Float
mediaTres a b c = (a + b + c)/3
```

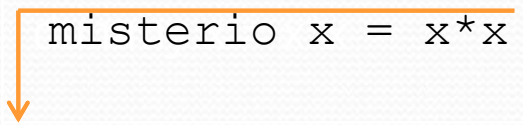
Regras de nomeação

- Haskell é *case sensitive*
  - Nomes de tipos começam com maiúsculas
    - Int, Integer, Char, ...
  - Nomes de variáveis e funções, com minúsculas
    - x, y, div, mod, abs, ...
  - Palavras reservadas não podem ser usadas: if, then, else, case,...

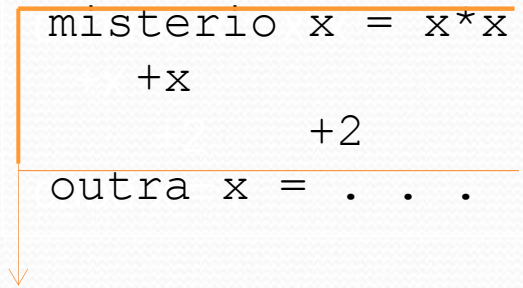
```
F :: int → int    -- erro na compilação
F X = X + 1       -- erro na compilação
```

Layout dos scripts: a regra do offside

Scripts são sensíveis à indentação



O primeiro caracter da definição abre uma caixa que contêm a definição



Tudo o que for digitado dentro da caixa faz parte da definição.
A caixa termina quando é encontrado um caracter em cima (ou à esquerda) da linha vertical.

A regra do offside

```
soma x = x+
x
```

Error ...: Syntax error in expression (unexpected ';')

O ';' marca o fim de uma definição. Assim, várias definições poderiam ser escritas numa só linha, por exemplo

```
soma :: Int->Int; soma x = x+ x; prod y = y*y
```

Em definições bem indentadas, seguindo a regra do *offside*, o ';' é implícito.

Funções

- Definidas através de equações
- Se recomenda explicitar o tipo
  - Exceto para pequenas definições auxiliares

```
quadX :: Int -> Int
quadX x = x*x
```

Avaliação de Funções

- Usando substituição, de maneira similar como na aritmética clássica, porém só realizando uma substituição a cada passo.
- A ordem de avaliação não interfere no resultado (transparência referencial).

```
quadX :: Int -> Int
quadX x = x*x
```

```
quadX (2+5) ↝ (2+5) * (2+5) ↝ 7 * (2+5) ↝ 7*7 ↝ 49
```

```
quadX (2+5) ↝  quadX 7 ↝ 7 * 7 ↝ 49
```

Guardas

```
maxi :: Int -> Int -> Int
maxi n m = if m >= n then m else n
```

```
maxi :: Int -> Int -> Int
maxi n m
      | m >=n      = m
      | otherwise = n
```

Avaliação

```
maxi :: Int -> Int -> Int
maxi n m = if m >= n then m else n
```

```
maxi (2+3) (4-1) ↝ maxi 5 (4-1) ↝ maxi 5 3 ↝
if 5 >= 3 then 5 else 3 ↝ if True then 5 else 3 ↝ 5
```

Avaliação com guardas

```
maxi :: Int -> Int -> Int
maxi n m
      | m >=n      = m
      | otherwise = n
```

```
maxi (2+3) (4-1)↷ maxi 5 (4-1) ↷
  maxi 5 3 ↷
?? 5 >= 3 ↷ True
  5
```

```
maxi (4-1) (2+3) ↷ maxi 3 (2+3) ↷
  maxi 3 5 ↷
?? 3 >= 5 ↷ False
?? otherwise ↷ True
  5
```

Exemplos de Funções

```
ehMinusculo :: Char -> Bool
ehMinusculo c = ('a' <= c) && (c <='z')

paraMaiusculo :: Char -> Char
paraMaiusculo c
    | ehMinusculo c = chr (ord c - ord 'a' + ord 'A')
    | otherwise = c
```

Praticando funções

- Elabore as seguintes funções e depois avalie–as em alguns casos:
  - Dados três números, determine se todos são diferentes.
  - Dados dois números, determine o menor deles.
  - Dados três números, determine o menor deles.
  - Dados três números, determine quantos estão acima da média dos três

dcomp departamento de computação

Operadores

- Operadores são funções com sintaxe especial
  - notação infixa
  - nomes formados com os símbolos:

    ! # $ % * + . / < = > ? @ \ ^ | : - ~

- Exemplo do uso de operadores em expressões:

```
(14+3)*5
```

Operadores e Funções

- A aplicação de uma função denota-se por justaposição

```
div   14   3

mod   (12+3)   4
```

- Convertendo operadores em funções e vice-versa

`(+) 2 3` equivale a `2 + 3`

`div 14 3` equivale a `14 `div` 3`

Operadores definidos pelo usuário

- Operadores definidos pelo usuário, usando os símbolos

  ! # $ % * + . / < = > ? @ \ ^ | : - ~

- Associatividade e precedência definida pelo usuário: infixl, infix, infixr

```
infixl 7 &&&
(&&&) :: Int -> Int -> Int
a &&& b
      | a >= b     = a
      | otherwise = b
```

Precedência e associatividade

	esquerda	não associa	direita
9	!, !!		.
8			**, ^, ^^
7	*,/,`div`		
8	`mod`		
6	+, -		
5			:, ++
4		/=, <, <=, ==, >, >=, `elem`, `notElem`	
3			&&
2			||
1	>>, >>=		
0			$

Precedência da aplicação

- A aplicação de uma função a um ou mais argumentos sempre tem maior precedência que qualquer operador. Assim, por exemplo:

`f x+1` deve ser entendido como `(f x) + 1`

Caso deseje aplicar `f` a `x+1` deve usar-se parêntesis:

```
f (x+1)
```

dcomp departamento de computação

Enganos com literais negativos

```
negate -5
```

- Expressão com erro de tipos
- O certo é

```
negate (-5)
```

Exercícios de Fixação

- Usando guardas, defina funções para
  - Dados os coeficientes *a*, *b* e *c* de uma equação de segundo grau

$$ax^2 + bx + c = 0$$

defina duas funções para calcular as raízes menor e maior. Se a equação não tiver raízes, as funções deverão retornar error.

Exercícios de Fixação

- Crie dois operadores novos para:
  - Dado dois inteiros devolver o dobro da soma dos números
  - Dado dois inteiros devolver o quadrado do produto dos números
  - Dados dois booleanos devolver o ou exclusivo deles

Funções com casamento de padrões

argumentos podem ser "casados" com padrões

```
myNot :: Bool -> Bool
myNot True = False
myNot False = True
```

```
myExOr :: Bool -> Bool -> Bool
myExOr True x = not x
myNot False x = x
```

Casamento de padrões produzem *bindings* (vínculos)

```
> myExOr True False
True
```

dcomp departamento de computação

Funções com casamento de padrões

- Haskell também tem o tipo `String` que é uma cadeia de caracteres; ele não é um tipo básico. Ex: “Leila”
- Várias equações junto com padrões

```
lucky :: Int -> String
lucky 7 = “Você acertou, sortudo!”
lucky x = “Oops, tente outra vez!”
```

dcomp departamento de computação

Funções com casamento de padrões

- Haskell faz a avaliação assim
  - Tenta usar a primeira equação fazendo o casamento de padrão
  - Se o casamento falhar, tenta a próxima equação, e assim por diante

```
> lucky 7
“Você acertou, sortudo!”
> lucky 0
“Oops, tente outra vez!”
```

Exercício de Fixação

- Usando casamento de padrão crie uma função chamada `nAnd` , que dadas duas variáveis Booleanas, retorna `True` em todos os casos à exceção do caso em que as duas variáveis sejam verdadeiras.

Funções com casamento de padrões

Podemos usar "_" quando não interessa qual é uma parte de um padrão

```
lucky :: Int -> String
lucky 7 = "Você acertou, sortudo!"
lucky _ = "Oops, tente outra vez!"
```

Exercício de Fixação

Reescreva a função `nAnd` usando _ para casar vários padrões

Exercício de Fixação

Reescreva a função `nAnd` usando _ para casar vários padrões

```
nAnd :: Bool -> Bool -> Bool
nAnd True True = False
nAnd _ _ = True
```

```
> nAnd True True
False
> nAnd False True
True
```

dcomp departamento de computação

Funções com casamento de padrões

- Casamento de padrão ajuda muitas vezes a melhorar a legibilidade. Suponha a função:

```
escrevaMe :: Int -> String
escrevaMe 1 = "Um!"
escrevaMe 2 = "Dois!"
escrevaMe 3 = "Tres!"
escrevaMe 4 = "Quatro!"
escrevaMe 5 = "Cinco!"
escrevaMe _ = "Cansei!"
```

Tente agora escrever esta função em única equação usando `if then else` aninhados. A legibilidade não vai ser boa! Experimente!

dcomp departamento de computação

Funções com casamento de padrões

- Casamento de padrões pode falhar

```
charNome :: Char -> String
charNome 'a' = "Alberto"
charNome 'b' = "Beatriz"
charNome 'c' = "Carlos"
```

```
> charNome 'a'
"Alberto"
> charNome 'b'
"Beatriz"
> charNome 'h'
"*** Exception: tut.hs:(53,0)-(55,21): Non-exhaustive
patterns in function charNome
```

Para não termos resultados inesperados, cuidado é necessário para incluir padrões e equações que cubram TODAS as possibilidades.

Definições locais

- Código repetido
  - suscetível a erros
  - difícil manter
  - ineficiente
  - difícil de ler

```
imc :: Float -> Float -> String
imc peso altura
    | peso / altura ^ 2 < 18.5 = "Abaixo do peso"
    | peso / altura ^ 2 < 25.0 = "Peso normal"
    | peso / altura ^ 2 < 30.0 = "Sobrepeso"
    | otherwise                = "Obesidade"
```

dcomp departamento de computação

Definições locais

```
imc :: Float -> Float -> String
imc peso altura
    | razao < 18.5 = “Abaixo do peso”
    | razao < 25.0 = “Peso normal”
    | razao < 30.0 = “Sobrepeso”
    | otherwise    = “Obesidade”
    where razao = peso / altura ^ 2
```

```
imc :: Float -> Float -> String
imc peso altura
    | razao < magro  = "Abaixo do peso"
    | razao < normal = "Peso normal"
    | razao < gordo  = "Sobrepeso"
    | otherwise      = "Obesidade"
    where razao  = peso / altura ^ 2
          magro  = 18.5
          normal = 25.0
          gordo  = 30.0
```

dcomp departamento de computação

Escopo de Definições Locais

- Escopo = trecho de alcance de uma definição
  - Nomes definidos numa cláusula **where** são visíveis somente na definição da função
  - Parâmetros formais são também locais

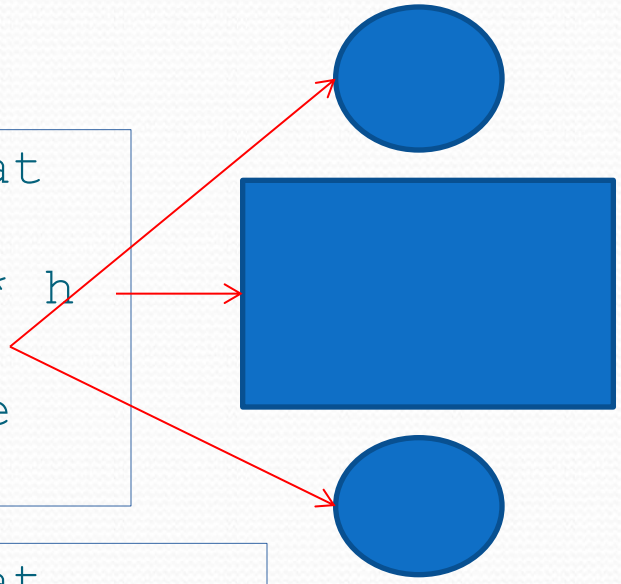
Expressões `let`

- Similar ao **where**, produz vínculos locais
- Mas, diferente do **where**, expressões **let** são expressões

```
>4 * (let a = 9 in a + 1) + 2
42
```

Expressões `let`

```
cilindro :: Float -> Float -> Float
cilindro r h =
    let areaLateral = 2 * pi * r * h
        areaBase    = pi * r^2
    in  areaLateral + 2 * areaBase
```



```
cilindro :: Float -> Float -> Float
cilindro r h = areaLateral + 2 * areaBase
    where areaLateral = 2 * pi * r * h
          areaBase    = pi * r^2
```

Neste caso, o mais comum é usar **where**

Expressões `let`

- Podem também ser usadas para introduzir funções

```
> let quadX x = x*x in quadX 4
16
```

- Pode usar ; para separar várias declarações locais

```
> let a=100; b=200; c=300 in a+b+c
600
```

Exercícios de Fixação

- Elabore uma função que determine se um inteiro é múltiplo de 3 e/ou de 5 ou de nenhum deles. A saída deve ser a mensagem “multiplo de 3 e de 5” “multiplo de 3” , “multiplo de 5” ou “não eh multiplo de 3 nem de 5”.
- Elabore uma função que dados dois inteiros representando os lados de uma forma geométrica retorne se esta forma é um quadrado ou retângulo.
- Elabore uma função que dados dois lados do retângulo calcula sua área.
- Usando a função acima, calcule a área de um cuboide de lados a, b e c.
- Usando expressões `let` e `where` refaça a questão anterior sem usar a função retângulo externamente à sua função.
- Usando casamento de padrão escreva uma função que dada sua cor preferida retorna seu significado, ou dá uma mensagem de consolação se não souber o significado. Você pode usar: branco-paz, amarelo-alegria, verde-esperança , azul - tranquilidade, vermelho – paixão e qualquer outra desculpa.