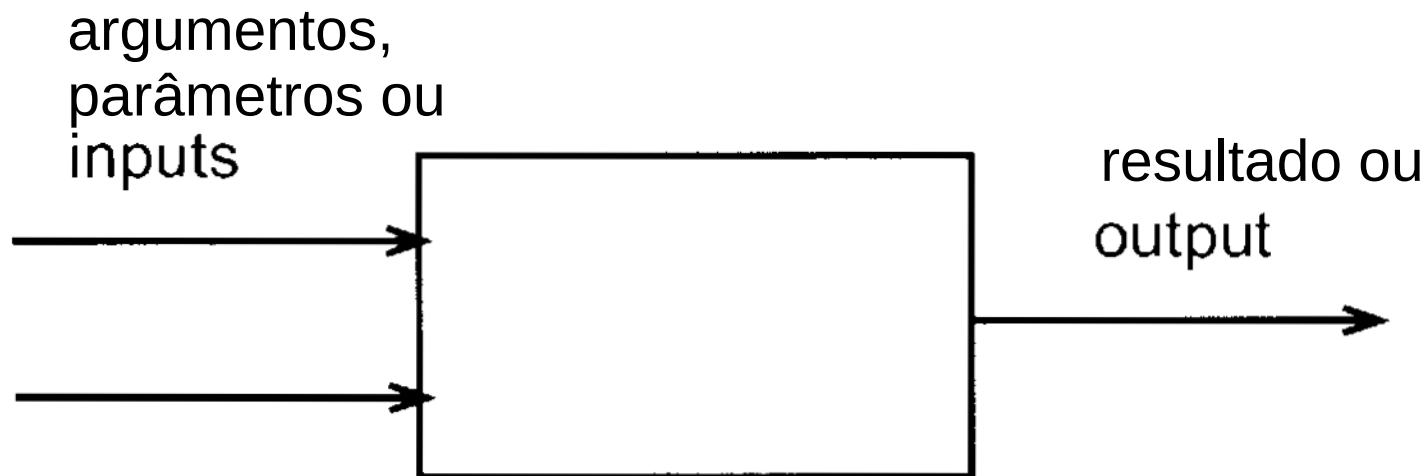


Tipos Básicos e Definição de Funções

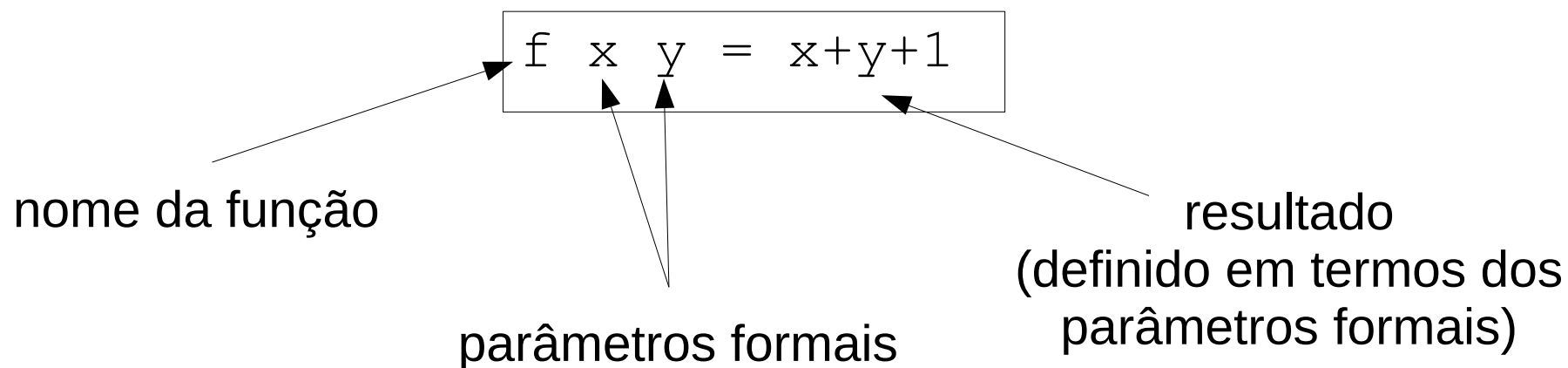
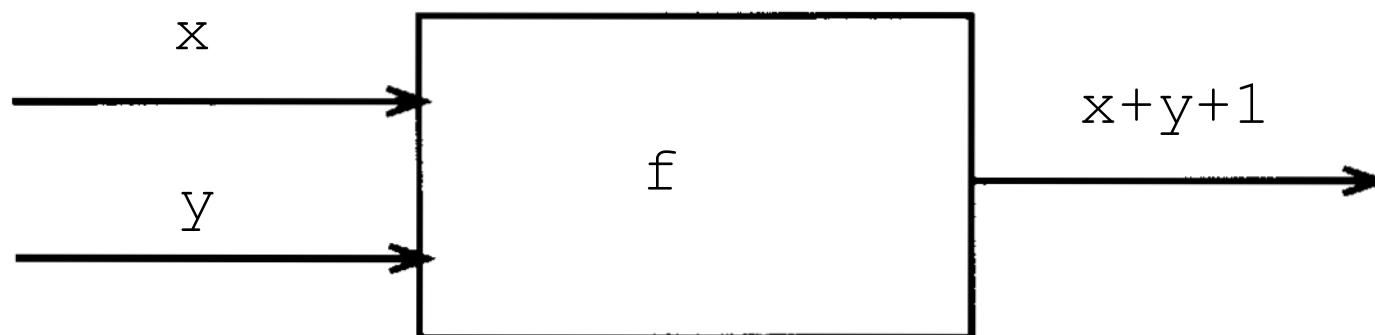
Programação Funcional
DCOMP-UFS

Função

- Relação binária tal que
- “Caixas pretas” que recebem valores de entrada e produzem um valor que depende dos valores da entrada



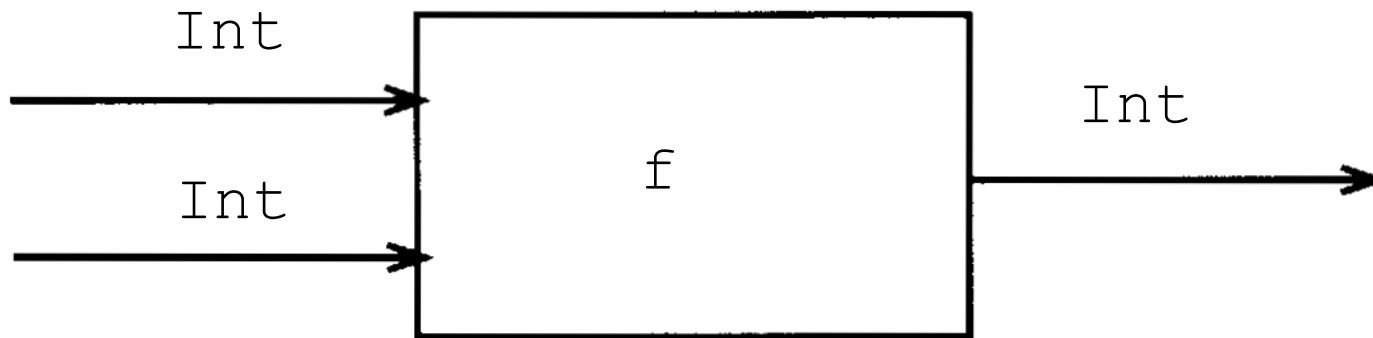
Definição de função



Tipo

- Funções recebem e devolvem valores/dados
- Dados são classificados em tipos
- Tipo
 - Conjunto de valores
 - Funções que operam uniformemente sobre os valores

Declaração de tipo



```
f :: Int -> Int -> Int
```

Em Haskell toda expressão têm tipo

- Restringe a usos coerentes
- Checagem estática (antes da execução)
 - Programas mais confiáveis (erros detectados cedo)
 - Programas mais legíveis
 - Programas mais eficientes
- Tipos podem ser explicitados pelo programador
 - Se não for explicitado, Haskell infere o tipo

Definição junto com a declaração de tipo

```
f :: Int -> Int -> Int  
f x y = x+y+1
```

Expressões

- O interpretador de Haskell (GHCi) permite avaliar expressões como

$(7 - 2) * 3$

- Expressões podem ser construídas chamando funções
- Notação de chamada por justaposição

$f \ (2 * 3) \ 5$

Regras de nomeação

- Haskell é “*case sensitive*”
 - Nomes de tipos começam com maiúsculas
 - Int, Integer, Char, ...
 - Nomes de variáveis e funções, com minúsculas
 - div, mod, ...

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

Classificação de Tipos

- Primitivos
 - valores atômicos
 - inteiros, caracteres, reais, booleanos, enumerados, ...
- Tipos compostos (ou estruturados)
 - tuplas, listas, ...

O tipo dos booleanos

- **Bool**
 - True, False
 - Operações: &&, ||, ==
 - Funções: not

- Tabelas de verdade

Tipos Inteiros

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

Sobrecarga

- Considere

```
a :: Int
b :: Int
c :: Integer
d :: Integer
... a+b... c+d ...
```

- As operações efetuadas ao avaliar $a+b$ e $c+d$ são diferentes
- **Sobrecarga**: um mesmo nome para diferentes operações ou funções
- Similarmente, os nomes das funções `succ`, `div`, `mod`, `abs`, `negate` são sobrecarregados

Caracteres

- **Char**

`'a', 'b', '\t', '\n', '\\', '\34'`

`<, <=, >, >=, ==, /=`

`- fromEnum :: Char -> Int`

`- toEnum :: Int -> Char`

Convertem de caractere para código numérico e vice-versa

Reais em ponto flutuante

- **Float, Double**

- 0, 4, -345, 0.31416, -23.12, 2.45e+2
- +, -, *, /, ^, **, <, <=, ...
- abs, acos, asin, atan, cos, sin, tan, exp, fromIntegral, abs, negate

Literais numéricos são sobrecarregados, por ex.

12

representa tanto o número `Int`, `Integer`, `Float` e `Double`

Haskell descobre qual é analisando o contexto, por exemplo, em

`12 + a`

se `a` é de tipo `Int`, o `12` também será de tipo `Int`

Sequências de caracteres

- **String**

Literais:

`"Maria"`

`"carro azul",`

`""`

`"gorila\thipopótamo\tgarça\n"`

Operações: `++` (concatenação)

Funções: `putStr, show, read`

- Não confunda `'a'` com `"a"`

Funções

- Se recomenda explicitar o tipo
 - Exceto para pequenas definições auxiliares

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

Como é feita a avaliação

- 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).

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

```
square (2+5)  $\rightsquigarrow$  (2+5) * (2+5)  $\rightsquigarrow$  7 * (2+5)  $\rightsquigarrow$  7*7  $\rightsquigarrow$  49
```

```
square (2+5)  $\rightsquigarrow$  square 7  $\rightsquigarrow$  7 * 7
```

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 $(f \ x) + 1$

Se se quer aplicar f a $x+1$ deve usar-se parêntesis, assim:

$f \ (x+1)$

Enganos com literais negativos

`f -5`

Expressão com erro de tipos

O certo é

`f (-5)`

Exemplos

```
eMinusc:: Char -> Bool
```

```
eMinusc c = 'a' <= c && c <= 'z'
```

```
eMaiusc:: Char -> Bool
```

```
eMaiusc c = 'A' <= c && c <= 'Z'
```

Guardas

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


Avaliação com guardas

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



```
nao :: Bool -> Bool
nao p
  | p          = False
  | otherwise = True
```

Definição com várias equações

```
lucky :: Int -> String
lucky 7 = "LUCKY NUMBER SEVEN!"
lucky x = "Sorry, you're out of luck, pal!"
```

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

2+4 é avaliado para analisar se a primeira equação pode ser usada

```
lucky (2+4) ≈
  lucky 6 ≈
    "Sorry, you're out of luck, pal!"
```

```
lucky :: Int -> String
lucky 7 = "LUCKY NUMBER SEVEN!"
lucky x = "Sorry, you're out of luck, pal!"
```

```
lucky (3+4) ~
  lucky 7 ~
    "LUCKY NUMBER SEVEN!"
```

Ordem das equações é importante

```
lucky :: Int -> String  
lucky x = "Sorry, you're out of luck, pal!"  
lucky 7 = "LUCKY NUMBER SEVEN!"
```

a primeira equação pode ser usada,
independentemente de qual é argumento

```
lucky (2+3) ≈  
    "Sorry, you're out of luck, pal!"
```

curinga (wild card)

```
lucky :: Int -> String  
lucky 7 = "LUCKY NUMBER SEVEN!"  
lucky x = "Sorry, you're out of luck, pal!"
```

```
lucky :: Int -> String  
lucky 7 = "LUCKY NUMBER SEVEN!"  
lucky _ = "Sorry, you're out of luck, pal!"
```

```
sayMe :: Int -> String
sayMe 1 = "One!"
sayMe 2 = "Two!"
sayMe 3 = "Three!"
sayMe 4 = "Four!"
sayMe 5 = "Five!"
sayMe _ = "Not between 1 and 5"
```



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

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

```
e :: Bool -> Bool -> Bool
e True True = True
e p q       = False
```

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

Exercício

- Defina a função `ouEx` que calcule o “ou exclusivo” entre dois valores booleanos
 - Dê uma versão usando guardas
 - Dê uma versão usando casamento de padrões (várias equações)
 - Dê uma versão usando uma fórmula booleana em uma única equação

```
ouEx :: Bool -> Bool -> Bool
ouEx p q ...
```

Exercícios

- Usando guardas, defina funções para
 - Calcular o menor de três números
 - Dados três números, calcular quantos estão acima da média
 - Dados os coeficientes a , b e c de uma equação de segundo grau

$$a x^2 + b x + c = 0$$

defina duas funções para calcular as raízes menor e maior.

Expressão condicional

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

Operadores e Funções

- Operadores são funções com sintaxe especial

- notação infixa

- nomes formados com os símbolos:

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

- Exemplo:

$(14 + 3) * 5$

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`, `mod`	
6	+, -		
5			:, ++
4		/=, <, <=, ==, >, >=, `elem`, `notElem`	
3			&&
2			
1		>>, >>=	
0			\$

Layout dos scripts: a regra do offside

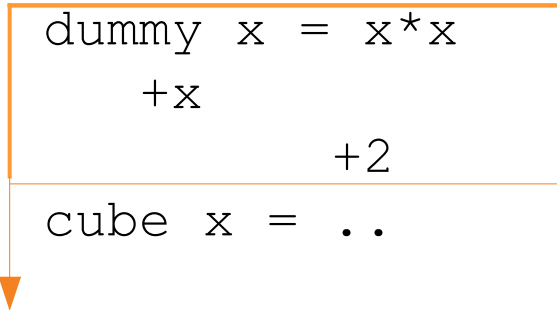
Scripts são sensíveis à indentação



```
dummy x = x*x
```

The diagram shows the code line 'dummy x = x*x'. An orange horizontal line is positioned above the text. From the left end of this line, an orange arrow points vertically downwards, passing through the first character 'd' of the word 'dummy'.

O primeiro caractere da definição abre uma “caixa” que contém a definição



```
dummy x = x*x
      +x
              +2
cube x = ..
```

The diagram shows a multi-line code block. The first line is 'dummy x = x*x'. The second line is '+x' with a tab stop. The third line is '+2' with two tab stops. The fourth line is 'cube x = ..'. An orange horizontal line is positioned above the first line. From the left end of this line, an orange arrow points vertically downwards, passing through the first character 'd' of the word 'dummy' on the first line and the first character 'c' of the word 'cube' on the fourth line.

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

A regra do offside

```
funny x = x+  
x
```

Error ...: Syntax error in expression (unexpected ‘;’)

De fato, ‘;’ marca o fim de uma definição. Assim, várias definições poderiam ser escritas numa só linha, por exemplo

```
funny::Int->Int; funny x = x+ x; happy y = y*y
```

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

Módulos e bibliotecas

- Para usar um módulo adicione
`import NomeModulo` no início do script
- `Data.Char` provê funções como `isLower`, `ord`,
`chr`, `isDigit`, ...

```
import Data.Char

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