Definição de funções

- A definição de funções faz-se através de uma sequência de equações da forma:

nome arg_1 arg_2 … arg_n = *expressão*

- O nome das funções começa sempre por letra minúscula ou *underscore*.
- Quando se define uma função podemos indicar o seu tipo. No entanto, isso não é obrigatório.
- O tipo de cada função é inferido automaticamente pelo compilador.
- O compilador infere o tipo mais geral que se pode associar à função. No entanto, é possível atribuir à função um tipo mais especifico.

```
swap :: (a,b) -> (b,a)
swap (x,y) = (y,x)
```

```
troca :: (Int,Char) -> (Char,Int)
troca (x,y) = (y,x)
```

- É boa prática de programação indicar o tipo das funções definidas nas scripts Haskell.

Exemplos

- Uma função de nome `fun` que recebe dois inteiros e, caso sejam ambos positivos, soma-os; caso contrário multiplica-os.

```
fun :: Int -> Int -> Int
fun x y = if x>0 && y>0 then x+y else x*y
```

- Uma função de nome `media` que recebe uma lista de inteiros e calcula a sua média aritmética.

```
media :: [Int] -> Int
media l = div (sum l) (length l)
```

Alternativamente poderíamos escrever:

```
media l = (sum l) `div` (length l)
```

Quando uma função é binária, podemos escreve-la de forma infixa (entre os seus argumentos) colocando o seu nome entre ` `

Exercícios

```
['a','b','c']
('a','b','c')
[(False,'0'),(True,'1')]
([False,True],['0','1'])
[tail, reverse, take 2]
```

Qual será o tipo destas expressões ?

```
second xs = head (tail xs)
pair x y = (x,y)
double x = x*2
palindrome xs = reverse xs == xs
twice f x = f (f x)
```

Qual será o tipo destas funções ?

Teste as suas respostas no GHCi.

Exercícios

```
['a','b','c']
('a','b','c')
[(False,'0'),(True,'1')]
([False,True],['0','1'])
[tail, reverse, take 2]
```

```
:: [Char]
:: (Char,Char,Char)
:: [(Bool,Char)]
:: ([Bool],[Char])
:: [[a]->[a]]
```

Exercícios

```
second xs = head (tail xs)
pair x y = (x,y)
double x = x*2
palindrome xs = reverse xs == xs
twice f x = f (f x)
```

```
:: [a] -> a
:: a -> b -> (a,b)
:: Num a => a -> a
:: Eq a => [a] -> Bool
:: (a->a) -> a -> a
```

Módulos

- Um programa Haskell está organizado em módulos.
- Cada módulo é uma coleção de definições num ambiente fechado.
- Um módulo pode exportar todas ou só algumas das suas definições. (...)

```
module Nome (...) where

...definições...
```

- Um módulo constitui um componente de software e dá a possibilidade de gerar bibliotecas de funções que podem ser reutilizadas em diversos programas.
- Para se utilizarem declarações feitas noutros módulos, que não o `Prelude`, é necessário primeiro fazer a sua importação através da instrução:

```
import Nome
```

Módulos

Exemplo: Muitas funções sobre caracteres estão definidas no módulo **`Data.Char`** .

```
module Exemplo where

import Data.Char

letra :: Int -> Char
letra n = if (n>=65 && n<=90) || (n>=97 && n<=122)
          then chr n
          else '-'

numero :: Int -> Char
numero n = if (n>=48 && n<=57)
           then chr n
           else '-'
```

`chr` é uma função do módulo `Data.Char` que dado o código ASCII de um caracter devolve o respectivo caracter.

```
> letra 100
'd'
> letra 5
'-'
> numero 3
'-'
> numero 55
'7'
```

ASCII (American Standard Code for Information Interchange)

Codificação standard dos caracteres de 8 bits baseada no alfabeto inglês.

A tabela vai de 0 a 127.

```
...
 9  - '\t'
10 - '\n'
...
32 - ' '
...
48 - '0'
49 - '1'
...
57 - '9'
...
65 - 'A'
66 - 'B'
...
90 - 'Z'
...
97 - 'a'
98 - 'b'
...
122 - 'z'
...
```

Comentários

O código Haskell pode ser comentado de duas formas:

`--` O texto que aparecer a seguir a `--` até ao final da linha é ignorado.

`{- … -}` O texto que estiver entre `{-` e `-}` não é avaliado pelo interpretador. Podem ser várias linhas.

```
module Exemplo where

import Data.Char

-- letra recebe um ASCII e devolve o caracter que lhe corresponde
-- se for uma letra; caso contrário dá '-'
letra :: Int -> Char
letra n = if (n>=65 && n<=90) || (n>=97 && n<=122)
          then chr n
          else '-'          -- porque não é uma letra

{- numero recebe um ASCII e devolve o caracter que lhe corresponde
se for um algarismo; caso contrário dá '-'    -}
numero :: Int -> Char
numero n = if (n>=48 && n<=57)
           then chr n
           else '-'
```

Tipos sinónimos

Em Haskell é possível renomear tipos através de declarações da forma

`type` *Nome* $p_1 \ldots p_n$ = *tipo*

Exemplos:

```
type Coordenada = (Float,Float)

distancia :: Coordenada -> Coordenada -> Float
distancia (x1,y1) (x2,y2) = sqrt ((x2-x1)^2+(y2-y1)^2)
```

```
type Triplo a = (a,a,a)

multri :: Triplo Int -> Int
multri (x,y,z) = x*y*z
```

Strings

O tipo **`String`** é um tipo sinónimo já definido no `Prelude`.

```
type String = [Char]
```

Os valores do tipo String podem ser escritos como sequências de caracteres entre aspas.

```
> ['o','l','a']
"ola"
> length "ola"
3
> reverse "ola"
"alo"
```

```
type Numero = Integer
type Nome = String
type Curso = String
type Aluno = (Numero,Nome,Curso)
type Turma = [Aluno]
```

Declarações locais

- Todas as declarações que vimos até aqui são globais. Ou seja, são visíveis no módulo onde estão.
- Muitas vezes é útil reduzir o âmbito de uma declaração, para tornar o código mais legível.
- O Haskell permite fazer
  - declarações locais a uma expressão, utilizando a construção **`let … in …`**

```
fun x = let v = x*x + x^10
        in x + v + 4*v
```

  - declarações locais a uma equação, utilizando a construção **`where …`**

```
exemplo x y = (a,b)
   where a = x+y
         b = sum [1..a]
```

Declarações locais

```
dis :: Coordenada -> Coordenada -> Float
dis (x1,y1) (x2,y2) = let a = (x2-x1)^2
                          b = (y2-y1)^2
                      in sqrt (a+b)
```

```
> dis (2,4.3) (-1,7.5)
4.386342
> dist (2,4.3) (-1,7.5)
4.386342
> a
error: Variable not in scope: a
```

```
dist :: Coordenada -> Coordenada -> Float
dist (x1,y1) (x2,y2) = sqrt (a+b)
  where a = (x2-x1)^2
        b = (y2-y1)^2
```

Também é possível definir localmente funções com argumentos .

Layout

O Haskell não necessita de marcas para delimitar as diversas declarações que constituem um programa. A identação do texto (isto é, a forma como o texto de uma definição está disposto), tem um significado preciso:

- Se uma linha começa mais à frente do que começou a linha anterior, então ela deve ser considerada como a continuação da linha anterior.
- Se uma linha começa na mesma coluna que a anterior, então elas são consideradas definições independentes.
- Se uma linha começa mais atrás do que a anterior, então essa linha não pertence à mesma lista de definições.

As declarações das funções `dis` e `dist` começam na mesma coluna.

```
dis :: Coordenada -> Coordenada -> Float
dis (x1,y1) (x2,y2) = let a = (x2-x1)^2
                          b = (y2-y1)^2
                      in sqrt (a+b)
```

As declarações de `a` e `b` dentro do `let-in` começam na mesma coluna.

`where` começa numa coluna mais à frente porque é a continuação da declaração da equação.

```
dist :: Coordenada -> Coordenada -> Float
dist (x1,y1) (x2,y2) = sqrt (a+b)
  where a = (x2-x1)^2
        b = (y2-y1)^2
```

As declarações de `a` e `b` dentro do `where` começam na mesma coluna.

Tipos algébricos

Em Haskell podemos definir novos tipos de dados através de declarações da forma

`data` *Nome* $p_1 \ldots p_n$ **=** *Construtor ...* | ... | *Construtor ...*

- Os construtores são os modos de construir valores do tipo que está a ser declarado.
- O nome dos construtores começa sempre por letra maiúscula.

Temos exemplos destas definições na biblioteca `Prelude`:

```
data Bool = False | True
```

```
data Maybe a = Nothing | Just a
```

```
> :type True
True :: Bool
> :type Nothing
Nothing :: Maybe a
> :type Just "ola"
Just "ola" :: Maybe [Char]
> :type Just True
Just True :: Maybe Bool
> :type Just
Just :: a -> Maybe a
```

Tipos algébricos

Exemplo: podemos definir um novo tipo para representar cores

```
data Cor = Amarelo | Verde | Vermelho | Azul
   deriving (Show)
```

Acrescentamos **`deriving (Show)`** para podermos visualizar os valores do novo tipo no interpretador.

e definir uma função que testa se uma cor é fria

```
fria :: Cor -> Bool
fria Verde = True
fria Azul = True
fria x = False
```

```
> :type Verde
Verde :: Cor
> fria Verde
True
> fria Amarelo
False
```

O GHCi procura de cima para baixo a equação que pode usar para calcular o valor da expressão `(fria Amarelo)`. A primeira que encontra é a 3ª equação.

Se invertermos a ordem das equações, qual será a resposta do GHCi ao avaliar a expressão `(fria verde)` ?

Tipos algébricos

Exemplo: podemos definir um novo tipo para representar pontos coloridos no plano cartesiano

```
data PontoC = PC Coordenada Cor
   deriving (Show)
```

```
> :type (PC (3.5,2.2) Azul)
(PC (3.5,2.2) Azul) :: PontoC
> :type (PC (-5,0.7) Verde)
(PC (-5,0.7) Verde) :: PontoC
> :type PC
PC :: Coordenada -> Cor -> PontoC
```

e definir uma função que calcula distância de um ponto colorido à origem do plano, assim

```
distOrigem :: PontoC -> Float
distOrigem (PC (x,y) c) = sqrt (x^2+y^2)
```

Ou então assim:

```
distOrigem :: PontoC -> Float
distOrigem (PC p c) = distancia p (0,0)
```

Padrões

- Um padrão é uma variável, uma constante, ou um construtor aplicado a outros padrões (isto é, um "esquema" de um valor atómico de um determinado tipo).
- Em Haskell, um padrão não pode ter variáveis repetidas (são padrões lineares).

```
distOrigem :: PontoC -> Float
distOrigem (PC (x,y) c) = sqrt (x^2+y^2)
```

`(PC (x,y) c)` é um padrão do tipo `PontoC`

- Ao definir funções colocamos como argumento um padrão do tipo do domínio da função.
- Quando a função é aplicada, o padrão que está no argumento da função é instanciado com o valor concreto, através de um mecanismo chamado de *pattern matching* (concordância de padrões) e as várias variáveis que compõem o padrão recebem um valor concreto.

```
> distOrigem (PC (2.5,3) Azul)
3.905125
```

O *pattern matching* é bem sucedido e `x=2.5`, `y=3` e `c=Azul`

Padrões

O mecanismo de pattern matching permite que possamos definir a mesma função de diferentes modos. Por exemplo,

```
distOrigem :: PontoC -> Float
distOrigem (PC p c) = distancia p (0,0)
```

`(PC p c)` é um padrão do tipo `PontoC`, sendo `p` um padrão do tipo `Coordenada` e `c` um padrão do tipo `Cor`.

Neste caso

```
> distOrigem (PC (2.5,3) Azul)
3.905125
```

O *pattern matching* é bem sucedido e `p=(2.5,3)` e `c=Azul`

Pattern matching

Recordemos a definição da função que testa se uma cor é fria

```
fria :: Cor -> Bool
fria Verde = True
fria Azul = True
fria x = False
```

Reparem que em todas as equações o argumento é sempre um padrão.

- Quando a função é aplicada a um valor concerto, o GHCi procura de cima para baixo a equação cujo lado esquerdo faz *match* e usa essa equação para calcular o resultado.
- Podemos ver um padrão como "uma forma" onde um valor concreto tem que encaixar.

```
> fria Azul
True

> fria Amarelo
False
```

O GHCi tenta usar a 1ª equação, mas não há *pattern matching* (pois `Verde≠Azul`). Depois tenta a 2ª equação e, como os padrões concordam (pois `Azul=Azul`), devolve o resultado de avaliar o lado direito da equação.

O GHCi tenta usar, sem sucesso, a 1ª e depois a 2ª equação. Depois aplica, com sucesso a 3ª equação, porque os padrões concordam (pois `x` é uma variável e `x=Amarelo`).

- Tudo isto faz com que a ordem em que aparecem as equações tenha influência no comportamento da função.

Maybe a

```
data Maybe a = Nothing | Just a
```

O tipo `Maybe a` pode ser usado para lidar com situações de excepção.

```
> div 5 0
*** Exception: divide by zero
```

```
myDiv :: Int -> Int -> Maybe Int
myDiv x y = if y==0
            then Nothing
            else Just (div x y)
```

```
> myDiv 5 0
Nothing
> myDiv 6 3
Just 2
```

Como poderemos tirar partido dos padrões para definir a função `myDiv` sem usar o if ?

```
myDiv x 0 = Nothing
myDiv x y = Just (div x y)
```

Maybe a

```
data Maybe a = Nothing | Just a
```

Exemplo: uma função para somar valores do tipo `Maybe Int` pode ser definida assim

```
myAdd :: Maybe Int -> Maybe Int -> Maybe Int
myAdd Nothing Nothing = Nothing
myAdd Nothing (Just y) = Nothing
myAdd (Just x) Nothing = Nothing
myAdd (Just x) (Just y) = Just (x+y)
```

Esta função pode ser definida de forma mais compacta. Como?

```
myAdd :: Maybe Int -> Maybe Int -> Maybe Int
myAdd (Just x) (Just y) = Just (x+y)
myAdd _ _ = Nothing
```

_ representa uma variável anónima. O GHCi gera automaticamente um nome novo para a variável. Costuma usar-se quando a variável não é utilizada no lado direito da equação.

Funções com guardas

Recorde a definição da função factorial

```
fact :: Integer -> Integer
fact 0 = 1
fact n = n * fact (n-1)
```

```
> fact 5
120
> fact 20
2432902008176640000
> fact (-1)
*** Exception: stack overflow
```

Porque a computação não termina e enche a *stack*.

Podemos definir `fact` usando uma guarda (condição) na segunda equação

```
fact :: Integer -> Integer
fact 0 = 1
fact n | n > 0 = n * fact (n-1)
```

```
> fact (-1)
*** Exception: Non-exhaustive patterns in function fact
```

A guarda é uma expressão Booleana. A equação só pode ser usada se a condição for verdadeira.

Funções com guardas

Uma definição alternativa para `fact` poderá ser

```
fact :: Integer -> Integer
fact 0 = 1
fact n | n > 0 = n * fact (n-1)
       | otherwise = error "Não está definida."
```

Aqui temos 2 equações com guardas.

A guarda `otherwise` corresponde a `True`.

A função `error :: String -> a` do `Prelude` permite alterar a mensagem de erro devolvida.

```
> fact (-1)
*** Exception: Não está definida.
```

Funções com guardas

- As expressões condicionais podem ser aninhadas.

```
sinal :: Int -> Int
sinal x = if x<0 then -1
          else if x==0 then 0
               else 1
```

- As equações guardadas podem ser usadas para tornar definições que envolvam if's aninhados mais fáceis de ler.

```
sinal x | x<0  = -1
        | x==0 = 0
        | otherwise = 1
```

- O uso de equações guardadas é também uma forma de contornar o facto de as expressões condicionais em Haskell terem obrigatoriamente o ramo `else`.

Operadores

- Operadores infixos (como o `+` , `*` , `&&` , ...) não são mais do que funções.
- Um operador infixo pode ser usado como uma função vulgar (i.e., usando notação prefixa) se estiver entre parêntesis.

```
> 3 + 2
5
> (+) 3 2
5
```

- Funções binárias podem ser usadas como um operador infixo, colocando o seu nome entre `` ` `` .

```
> div 10 3
3
> 10 `div` 3
3
```

- Podemos definir novos operadores infixos

```
(+>) :: Float -> Float -> Float
x +> y = x^2 + y
```

e indicar a prioridade e a associatividade através de declarações

```
infixl num op
infixr num op
infix  num op
```

Listas

- As listas são sequências de tamanho variável de elementos do mesmo tipo.
- As listas podem ser representadas colocando os seus elementos, separados por vírgulas, entre parêntesis rectos. Mas isso é açúcar sintáctico.

```
[1,2,3] :: [Int]
```

- Na realidade as listas são um tipo algébrico, cujos elementos são construídos à custa dos seguintes construtores:

```
[] :: [a]
(:) :: a -> [a] -> [a]
```

`[]` representa a lista vazia.

`(:)` é o constructor infixo que recebe um elemento e uma lista, e acrescenta o elemento à cabeça da lista (isto é, do lado esquerdo da lista) .
Nota: `(:)` é associativo à direita.

```
[1,2,3] = 1:[2,3] = 1:2:[3] = 1:2:3:[]
```

```
> 1:2:3:[]
[1,2,3]
> (2,3):(0,-1):[]
[(2,3),(0,-1)]
> 'B':"om dia!"
"Bom dia!"
```

Funções simples sobre listas

- **`head`** dá o primeiro elemento de uma lista não vazia, isto é, a cabeça da lista.

```
head :: [a] -> a
head (x:xs) = x
```

`(x:xs)` é um padrão que representa uma lista com pelo menos um elemento. `x` é o primeiro elemento da lista e `xs` é a restante lista.

Um padrão que é argumento de uma função tem que estar **entre parêntesis**, excepto se for uma variável ou uma constante atómica.

Pattern matching

```
> head [1,2,3]
1
> head [10,20,30,40,50]
10
> head []
*** Exception: Prelude.head: empty list
```

`x = 1 , xs = [2,3]`

`x = 10 , xs = [20,30,40,50]`

Não há *pattern matching*

Funções simples sobre listas

- **`tail`** retira o primeiro elemento de uma lista não vazia, isto é, dá a cauda da lista.

```
tail :: [a] -> [a]
tail (x:xs) = xs
```

Pattern matching

```
> tail [1,2,3]
[2,3]
> tail [10,20,30,40,50]
[20,30,40,50]
> tail []
*** Exception: tail: empty list
```

`x = 1 , xs = [2,3]`

`x = 10 , xs = [20,30,40,50]`

Não há *pattern matching*

Funções simples sobre listas

- **`null`** testa se uma lista é vazia.

```
null :: [a] -> Bool
null [] = True
null (x:xs) = False
```

Pattern matching

```
> null [1,2,3]
False
> null []
True
```

Falha o pattern matching na 1ª equação.
Usa a 2ª equação com sucesso `x=1, xs=[2,3]`

Usa a primeira equação com sucesso

Funções simples sobre listas

Exemplo: a função que soma os 3 primeiros elementos de uma lista de inteiros pode ser definida assim

```
soma3 :: [Int] -> Int
soma3 l | length l <= 3 = sum l
        | otherwise = sum (take 3 l)
```

Esta é uma definição pouco eficiente, pois temos que calcular o comprimento da lista, para depois somar apenas os seus 3 primeiros elementos.

Como poderemos definir essa função sem utilizar funções auxiliares e tirando partido do mecanismo de *pattern matching*?

```
soma3 :: [Int] -> Int
soma3 (x:y:z:t) = x+y+z
soma3 (x:y:t) = x+y
soma3 (x:t) = x
soma3 [] = 0
```

Note que a ordem relativa das 3 primeiras equações tem que ser esta.

O que acontece se passarmos a 3ª equação para 1º lugar?

Funções simples sobre listas

Outra alternativa para a função `soma3` pode ser assim

```
soma3 :: [Int] -> Int
soma3 [] = 0
soma3 [x] = x
soma3 [x,y] = x+y
soma3 l = sum (take 3 l)
```

`[x]` é uma lista com exactamente 1 elemento. `[x]==(x:[])`
`[x,y]` é uma lista com exactamente 2 elementos. `[x,y]==(x:y:[])`
`l` é uma lista qualquer mas a equação só irá ser usada com listas com mais de dois elementos, dada a sua posição relativa.

Não confundir os padrões aqui usados com os usados na versão anterior

```
soma3 :: [Int] -> Int
soma3 (x:y:z:t) = x+y+z
soma3 (x:y:t) = x+y
soma3 (x:t) = x
soma3 [] = 0
```

`(x:y:z:t)` é uma lista com pelo menos 3 elementos.
`(x:y:t)` é uma lista com pelo menos 2 elementos.
`(x:t)` é uma lista com pelo menos 1 elemento.