Layout

Ao contrário de quase todas as linguagens de programação, o Haskell não necessita de marcas para delimitar as diversas declarações que constituem um programa.

Em Haskell a *identação do texto* (isto é, a forma como o texto de uma definição está disposto), tem um significado bem preciso.

Regras fundamentais:

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

Ou seja: *definições do mesmo género devem começar na mesma coluna*

Exemplo:

```
exemplo :: Float -> Float -> Float
exemplo x 0 = x
exemplo x y = let a = x*y
                  b = if (x>=y) then x/y
                                else y*x
                  c = a-b
              in (a+b)*c
```


Módulos

Um programa Haskell está organizado em módulos.

Cada **módulo** é uma colecção de funções e tipos de dados, definidos num ambiente fechado.

Um módulo pode exportar todas ou só algumas das suas definições. (...)

module *Nome* (*nomes_a_exportar*) **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 Haskell.

Exemplo: Muitas funções sobre caracteres estão definidas no módulo Char do ghc.

Para se utilizarem declarações feitas noutros módulos é necessário primeiro fazer a sua importação.

import *Nome_do_módulo*

Normalmente, cada módulo está armazenado num ficheiro com o mesmo nome do módulo.

Exemplo.hs

```
module Exemplo where

import 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 ' '
```


Para criar programas *executáveis* o compilador Haskell precisa de um módulo Main com uma função main.

module Main where

... declarações ...

main = ...

... declarações ...

`main` tem de ser de tipo IO *(... falaremos disto mais tarde)*

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 parentesis.

Exemplo: `(+) 2 3` é equivalente a `2+3`

Note que `(+) :: Int -> Int -> Int`

Podem-se definir novos operadores infixos.

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

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

Exemplo: `mod :: Int -> Int -> Int`

`` 3 `mod` 2 `` é equivalente a `mod 3 2`

Cada operador tem uma prioridade e uma associatividade estipulada.

Isto faz com que seja possível evitar alguns parentesis.

Exemplo: `x + y + z` é equivalente a `(x + y) + z`

`x + 3 * y` é equivalente a `x + (3 * y)`

A aplicação de funções tem prioridade máxima e é associativa à esquerda.

Exemplo: `f x * y` é equivalente a `(f x) * y`

É possível indicar a prioridade e a associatividade de novos operadores através de declarações.

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


Funções com Guardas

Em Haskell é possível definir funções com alternativas usando **guardas.**

Uma guarda é uma expressão booleana. Se o seu valor for `True` a equação correspondente é usada na redução (senão tenta a seguinte).

Exemplos:

```
sig x y = if x > y then 1
                   else if x < y then -1
                                 else 0
```

é equivalente a

```
sig x y | x > y  = 1
        | x < y  = -1
        | x == y = 0
```

ou a

```
sig x y
    | x > y     = 1
    | x < y     = -1
    | otherwise = 0
```

`otherwise` é equivalente a `True`.

Exemplo: Raizes reais do polinómio $a\,x^2 + b\,x + c$

```
raizes :: (Double,Double,Double) -> (Double,Double)
raizes (a,b,c) = (r1,r2)
  where r1 = (-b + r) / (2*a)
        r2 = (-b - r) / (2*a)
        d = b^2 - 4*a*c
        r | d >= 0 = sqrt d
          | d < 0 = error "raizes imaginarias"
```

error é uma função pré-definida que permite indicar a mensagem de erro devolvida pelo interpretador. Repare no seu tipo

```
error :: String -> a
```

```
> raizes (2,10,3)
(-0.320550528229663,-4.6794494717703365)

> raizes (2,3,4)
*** Exception:  raizes imaginarias
```

Listas

`[T]` é o tipo das listas cujos elementos são todos do tipo `T` -- *listas homogéneas* .

```
[3.5^2, 4*7.1, 9+0.5] :: [Float]
[(253,"Braga"), (22,"Porto"), (21,"Lisboa")] :: [(Int,String)]
[[1,2,3], [1,4], [7,8,9]] :: [[Integer]]
```

Na realidade, as listas são construidas à custa de dois construtores primitivos:

— a lista vazia `[]`

— o construtor `(:)`, que é um operador infixo que dado um elemento `x` de tipo `a` e uma lista `l` de tipo `[a]`, constroi uma nova lista com `x` na 1ª posição seguida de `l`.

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

`[1,2,3]` é uma abreviatura de `1:(2:(3:[]))` que é igual a `1:2:3:[]` porque `(:)` é associativa à esquerda.

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

Os padrões do tipo lista são expressões envolvendo apenas os construtores `:` e `[]` (*entre parentesis*), ou a representação abreviada de listas.

```
head (x:xs) = x
```

```
tail (x:xs) = xs
```

```
null [] = True
null (x:xs) = False
```

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

Qual o tipo destas funções ?

As funções são totais ou parciais?

```
> head [3,4,5,6]
3
> tail "HASKELL"
"ASKELL"
> head []
*** exception
> null [3.4, 6.5, -5.5]
False
> soma3 [5,7]
13
```

Em `soma3` a ordem das equações é importante ? Porquê ?

Listas por Compreensão

Inspirada na forma de definir conjuntos por compreensão em linguagem matemática, a linguagem Haskell tem também mecanismos para definir listas por compreensão.

$\{ 2x \mid x \in \{10,3,7,2\} \}$ `[ 2*x | x <- [10,3,7,2] ] = [20,6,14,4]`

$\{ n \mid n \in \{9,8,-2,-10,3\} \wedge 0 \leq n+2 \leq 10 \}$

`[ n | n <- [9,8,-2,-10,3] , 0<=n+2, n+2<=10 ] = [8,-2,3]`

$\{4,7, \ldots , 19\}$ `[4,7..19] = [4,7,10,13,16,19]`

`[1..7] = [1,2,3,4,5,6,7]`

$\{ (x,y) \mid x \in \{3,4,5\} \wedge y \in \{9,10\} \}$ `[ (x,y) | x <- [3,4,5], y <- [9,10] ]`
`= [(3,9),(3,10),(4,9),(4,10),(5,9),(5,10)]`

Listas infinitas

$\{5,10, \ldots \}$ `[5,10..] = [5,10,15,20,25,30,35,40,45,50,55,...`

$\{ x^3 \mid x \in \mathbb{N} \wedge \text{par}(x) \}$ `[ x^3 | x <- [0..], even x ] = [0,8,46,216,...`

Recorrência

Como definir a função que calcula o comprimento de uma lista ?

Temos dois casos:

— Se a lista fôr vazia o seu comprimento é zero.

— Se a lista não fôr vazia o seu comprimento é um mais o comprimento da cauda da lista.

```
length [] = 0
length (x:xs) = 1 + length xs
```

Esta função é **recursiva** uma vez que se invoca a si própria (aplicada à cauda da lista).

A função termina uma vez que as invocações recursivas são feitas sobre listas cada vez mais curtas, e vai chegar ao ponto em que a lista é vazia.

```
length [1,2,3] = length (1:[2,3])  ⇒  1 + length [2,3]
                                   ⇒  1 + 1 + length [3]
                                   ⇒  1 + 1 + 1 + length []
                                   ⇒  1 + 1 + 1 + 0
                                   ⇒  3
```

Em linguagens funcionais, a recorrência é a forma de obter ciclos.

Mais alguns exemplos de funções já definidas no módulo Prelude:

```
sum [] = 0
sum (x:xs) = x + sum xs
```

Qual o tipo destas funções ?

São totais ou parciais ?

Podemos trocar a ordem das equações ?

```
last [x] = x
last (_:xs) = last xs
```

```
elem x [] = False
elem x (y:ys) | x == y    = True
              | otherwise = elem x ys
```

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


Considere a função `zip` já definida no Perlude:

```
zip [] [] = []
zip [] (y:ys) = []
zip (x:xs) [] = []
zip (x:xs) (y:ys) = (x,y) : (zip xs ys)
```

Qual o seu tipo ?   É total ou parcial ?
Podemos trocar a ordem das equações ?
Podemos dispensar alguma equação ?

Exercícios:

Indique todos os passos de redução envolvidos no cálculo da expressão:

`zip [1,2] “LESI”`

Defina a função que faz o *zip* de 3 listas.

Defina a função   `unzip :: [(a,b)] -> ([a],[b])`

Acumuladores

Considere a definição da função factorial.

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

O cálculo da factorial de um número positivo `n` é feito multiplicando `n` pelo factorial de `(n-1)`.

A multiplicação fica *em suspenso* até que o valor de `fact (n-1)` seja sintetizado.

```
fact 3  ⇒  3*(fact 2)  ⇒  3*(2*(fact 1))  ⇒  3*(2*(1*(fact 0)))
        ⇒  3*(2*(1*1)) ⇒  6
```

Uma outra estratégia para resolver o mesmo problema, consiste em definir uma função auxiliar com um parametro extra que serve para ir guardando os resultados parciais – a este parametro extra chama-se **acumulador**.

```
fact n | n >=0 = factAc 1 n
  where factAc ac 0 = ac
        factAc ac n = factAc (ac*n) (n-1)
```

```
fact 3  ⇒  factAc 1 3  ⇒  factAc (1*3) 2  ⇒ factAc (1*3*2) 1
        ⇒  factAc (1*2*3*1) 0  ⇒  1*2*3*1  ⇒  6
```


Dependendo do problema a resolver, o uso de acumuladores pode ou não trazer vantagens.

Por vezes, pode ser a forma mais natural de resolver um problema.

Exemplo:

Considere as duas versões da função que faz o cálculo do valor máximo de uma lista.

Qual lhe parece mais natural ?

```
maximum [x] = x
maximum (x:y:xs) | x > y     = maximum (x:ys)
                 | otherwise = maximum (y:xs)
```

```
maximo (x:xs) = maxAc x xs
  where maxAc ac [] = ac
        maxAc ac (y:ys) = if y>ac then maxAc y ys
                                  else maxAc ac ys
```

Em `maximo` o acumulador guarda o valor máximo encontrado até ao momento.

Em `maximum` a cabeça da lista está a funcionar como acumulador.

Considere a função que inverte uma lista.

```
reverse [] = []
reverse (x:xs) = (reverse xs) ++ [x]
```

```
reverse [1,2,3]  ⇒ (reverse [2,3])++[1] ⇒ ((reverse [3])++[2])++[1]
          ⇒ (((reverse [])++[3])++[2])++[1]  ⇒ ((([]++[3])++[2])++[1]
          ⇒ ([3]++[2])++[1] ⇒ (3:([]++[2]))++[1]  ⇒ (3:[2])++[1]
          ⇒ 3:([2]++[1]) ⇒ 3:(2:([]++[1])) ⇒  3:2:[1] =  [3,2,1]
```

Este é um exemplo típico de uma função que implementada com um acumulador é muito mais eficiente.

```
reverse l = revAc [] l
  where revAc ac [] = ac
        revAc ac (x:xs) = revAc (x:ac) xs
```

```
reverse [1,2,3] ⇒  revAc [] [1,2,3] ⇒  revAc [1] [2,3]
                ⇒  revAc [2,1] [3]  ⇒  revAc [3,2,1] []  ⇒ [3,2,1]
```


Padrões sobre números naturais.

O Haskell aceita expressões da forma **(** *variável* **+** número_natural **)** como um padrão sobre números naturais.

Exemplos:

```
fact 0 = 1
fact (n+1) = (n+1) * (fact n)
```

```
decTres (x+3) = x
```

```
> fact 4
24
> fact (-2)
*** Exception: Non-exhaustive patterns in function fact
```

```
> decTres 5
2
> decTres 10
7
> decTres 2
*** Exception: Non-exhaustive ...
```

Atenção:
expressões, como por exemplo,

`(n*5), (x-4) ou (2+n)`

não são padrões !

Mais alguma funções sobre listas pré-definidas no `Prelude`.

```
(x:_) !! 0  = x
(_:xs) !! (n+1) = xs !! n
```

O que fazem estas funções ?

```
init [x] = []
init (x:xs) = x : init xs
```

Qual o seu tipo ?

```
take 0 _ = []
take _ [] = []
take n (x:xs) = x : take (n-1) xs
```

```
drop 0 xs = xs
drop _  [] = []
drop n (_:xs) = drop (n-1) xs
```


Funções e listas por compreensão

Pedem-se usar listas por compreensão na definição de funções.

Exemplo: Máximo divisor comum de dois números.

```
divisores n = [ x | x <- [1..n], (n `mod` x) == 0 ]
```

```
divisoresComuns x y = [ n | n <- divisores x, (y `mod` n) == 0 ]
```

```
mdc n m = maximum (divisoresComuns n m)
```