Programação Funcional

1º Ano

Maria João Frade - Dep. Informática, UM

O que é a programação funcional ?

- É um estilo de programação em que o mecanismo básico de computação é a aplicação de funções a argumentos.
- Uma linguagem de programação que suporte e encoraje esta forma de programação diz-se funcional.
- É um estilo de programação declarativo (em que um programa é um conjunto de declarações que descrevem a relação entre input e output), ao invés de um estilo imperativo (em que o programa é uma sequência de instruções que vai alterando o estado, o valor das variáveis).

Exemplo da função factorial

0! = 1
n! = n . (n-1)!

Haskell (uma linguagem declarativa)

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

As equações que são usadas na definição de `fact` são equações matemáticas.
Elas indicam que o lado esquerdo e o lado direito do `=` têm o mesmo valor.
O valor dos identificadores é imutável.

C (uma linguagem imperativa)

```
int factorial(int n)
{ int i, r;
  i = 1;
  r = 1;
  while (i<=n) {
     r = r*i;
     i = i+1; }
  return r;
}
```

Isto é muito diferente do uso do `=` nas linguagens imperativas como o C. Neste caso, o valor dos identificadores é mutável. Por exemplo, a instrução `i = i+1` representa uma atribuição (o valor anterior de `i` é destruído, e o novo valor passa a ser o anterior mais 1). Portanto, o valor de `i` é alterado.

Programa resumido

Esta UC corresponde a uma introdução ao paradigma funcional de programação, tendo por base a linguagem programação **Haskell** (uma linguagem puramente funcional).

1. Aspectos básicos da linguagem Haskell: Valores, expressões e tipos. O mecanismo de avaliação. Inferência de tipos. Definições multi-clausais de funções. Polimorfismo.
2. Listas. Funções recursivas sobre listas. Modelação de problemas usando listas.
3. Algoritmos de ordenação de listas: *insertion sort*, *quick sort* e *merge sort*.
4. Ordem superior: Padrões de computação. Programação com funções de ordem superior.
5. Tipos algébricos: Definição de novos tipos e sua utilização na modelação de problemas.
6. Árvores: Árvores binárias, árvores de procura, árvores irregulares e algoritmos associados.
7. Classes: O mecanismo de classes no tratamento do polimorfismo e da sobrecarga de funções.
8. IO: O tratamento puramente funcional do input/output. O monade IO.

Resultados de aprendizagem

- Resolver problemas de programação decompondo-os em problemas mais pequenos.
- Desenvolver e implementar algoritmos recursivos sobre listas e sobre árvores.
- Desenvolver programas tirando partido da utilização das funções de ordem superior.
- Definir tipos algébricos enquadrá-los na hierarquia de classes e programar com esses tipos.
- Escrever programas interativos.

Método de avaliação

Nota Final = (Nota do 1º teste) x (Nota do 2º teste)

- **1º teste:** uma questão selecionada aleatoriamente de um conjunto de questões simples, previamente divulgado. (Notas 0 ou 1)
- **2º teste:** prova escrita sobre toda a matéria. (Notas de 0 a 20)

Datas previstas para as avaliações

1º teste: 3 de Novembro
2º teste: 9 de Janeiro
Exame de recurso: 30 de Janeiro

Bibliografia

- Fundamentos da Computação. Livro II: Programação Funcional.
  José Manuel Valença e José Bernardo Barros. Universidade Aberta.
- Programming in Haskell. Graham Hutton. Cambridge University Press, 2016.
- Haskell: the craft of functional programming. Simon Thompson. Addison-Wesley.
- www.haskell.org/documentation
- Slides das aulas teóricas e fichas práticas: elearning.uminho.pt

Características das linguagens funcionais

- O mecanismo básico de programação é a definição e aplicação de funções.
- Funções são entidades de 1ª classe, isto é, podem ser usadas como qualquer outro objecto: passadas como parâmetro, devolvidas como resultado, ou mesmo armazenadas em estruturas de dados.
- Grande flexibilidade, capacidade de abstração e modularização do processamento de dados.
- Os programas são concisos, fáceis de manter e rápidos de desenvolver.

Cronologia

1930's - *Lambda calculus*: uma teoria matemática the funções. *(Alonso Church, Haskell Curry)*

1950's - Lisp: a 1ª ling. prog. funcional, sem tipos, impura. *(John McCarthy)*

1960's - ISWIN: a 1ª ling. prog. funcional pura. *(Peter Landin)*

1970's - FP: ênfase nas funções de ordem superior e no raciocínio sobre programas. *(John Backus)*
ML: a 1ª ling. funcional moderna, com polimorfismo e inferência de tipos. *(Robin Milner)*

1980's - Miranda: ling. funcional com *lazy evaluation,* polimorfismo e inferência de tipos. *(David Turner)*

1990's - Haskell: ling. funcional pura, *lazy*, com um sistema de tipos extremamente evoluído, criada por um comité de académicos.

2000's - Publicação do Haskell Report: a 1ª versão estável da linguagem, actualizada em 2010.

2010's - Haskell Platform: distribuição standard do GHC (Glasgow Haskell Compiler), que inclui bibliotecas e ferramentas de desenvolvimento.

Haskell

Haskell is a general purpose, purely functional programming language incorporating many recent innovations in programming language design. Haskell provides higher-order functions, non-strict semantics, static polymorphic typing, user-defined algebraic datatypes, pattern-matching, list comprehensions, a module system, a monadic IO system, and a rich set of primitive datatypes, including lists, arrays, arbitrary and fixed precision integers, and floating-point numbers. Haskell is both the culmination and solidification of many years of research on non-strict functional languages.

(The Haskell 2010 Report)

www.haskell.org

Haskell with batteries included

Contém o compilador de Haskell **GHC**, que vamos usar.

Glasgow Haskell Compiler

- Principal compilador de Haskell da actualidade.
- Usado na indústria.
- Inclui um compilador e um interpretador de Haskell:
  - **GHC** - o compilador que a partir do programa Haskell cria código executável.
  - **GHCi** - o interpretador que actua como uma "máquina de calcular". Tem uma natureza interactiva adequada ao desenvolvimento passo a passo de um programa. É o que usaremos nas aulas.

O ciclo de funcionamento do interpretador é o seguinte:

***lê** uma expressão, **calcula** o seu valor e **apresenta** o resultado*

O interpretador GHCi

O interpretador arranca, a partir de um terminal, com o comando **ghci**

```
$ ghci
GHCi, version … : http://www.haskell.org/ghc/. :? for help
Prelude>
```

- O *prompt* **>** indica que o GHCi está pronto para avaliar.
- **Prelude** é o nome da biblioteca que é carregada, por omissão, no arranque do GHCi e que disponibiliza uma vasta lista de funções.

```
Prelude> 5+3*2
11
Prelude> sqrt 9
3.0
```

A biblioteca `Prelude`

O `Prelude` é a biblioteca Haskell que contém as declarações de tipos, funções e classes que constituem o núcleo central da linguagem Haskell. É sempre carregada por omissão.

Por exemplo, tem muitas funções sobre listas:

```
> length [4,2,6,3,1]
5
> head [4,2,6,3,1]
4
> tail [4,2,6,3,1]
[2,6,3,1]
> reverse [4,2,6,3,1]
[1,3,6,2,4]
> last [1..5]
5
> sum [4,2,6,3,1]
16
```

```
> product [1..5]
120
> [1,2,3] ++ [4,5]
[1,2,3,4,5]
> head (tail [1..5])
2
> take 2 [3,4,7,1,8]
[3,4]
> drop 2 [3,4,7,1,8]
[7,1,8]
> length [4,2,1] + head [7,5]
10
```


Aplicação de funções

A notação usada em Haskell para a aplicação de funções difere da notação matemática tradicional.

- Na matemática, a aplicação de funções é denotada usando parêntesis e multiplicação denotada por um espaço.
- Em Haskell, a aplicação de funções é denotada por um espaço e multiplicação denotada por *.

Notação matemática	Notação Haskell
f(a,b) + c d	`f a b + c*d`

Em Haskell a aplicações de funções tem prioridade máxima sobre todos os outros operadores.

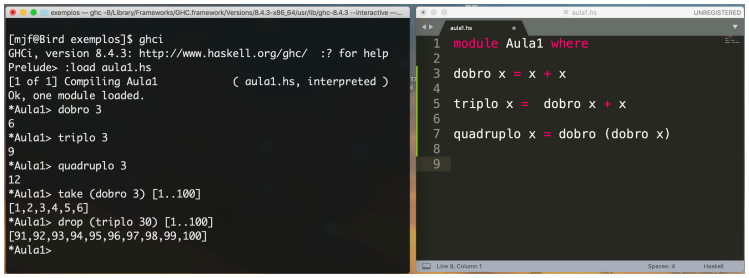
`f a + b` significa `(f a) + b`

Aplicação de funções

Notação matemática	Notação Haskell
f(x)	`f x`
f(x,y)	`f x y`
f(g(x))	`f (g x)`
f(x,g(x))	`f x (g x)`
f(a) + b	`f a + b`

Haskell scripts

- Um programa Haskell é constituído por um, ou mais, ficheiros de texto que contêm as definições das novas funções, tipos e classes usados na resolução de um dado problema.
- A esses ficheiros Haskell costumam-se chamar *scripts*, pelo que o nome dos ficheiros Haskell termina normalmente com **.hs** (de Haskell script).
- No desenvolvimento de um programa Haskell é útil manter duas janelas abertas: uma com o editor de texto onde se vai desenvolvendo o programa, e outra com o GHCi para ir testando as funções que se vão definindo.



Haskell scripts

Mantendo o GHCi aberto podemos acrescentar mais definições ao ficheiro aula1.hs e depois recarrega-lo no GHCi para as testar.

Por exemplo, podemos acrescentar ao ficheiro a definição da função factorial

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

E depois recarrega-lo no GHCi

```
*Aula1> :reload
[1 of 1] Compiling Aula1     (aula1.ha. interpreted)
Ok, one module loaded
*Aula1> fact 5
120
*Aula1>
```

Repare na mudança de nome do prompt para *Aula1> que é o nome do módulo que está no ficheiro aula1.hs. Neste momento tem disponíveis no interpretador todas as funções do Prelude e do módulo Aula1.

Alguns comandos do GHCi

Comando	
`:?`	Mostra todos os comandos disponíveis
`:load` *nome*	Carrega no GHCi o ficheiro *nome*
`:reload`	Carrega de novo o ficheiro corrente
`:type` *expressão*	Indica o tipo de uma expressão
`:quit`	Sai do GHCi

Pode usar apenas a primeira letra do comando. Por exemplo, `:l aula1.hs`

Valores, expressões e tipos

- Os valores são as entidades básicas da linguagem Haskell. São os elementos atómicos.
- Uma expressão ou é um valor ou resulta de aplicar funções a expressões.
- O interpretador atua como uma calculadora: *lê uma expressão, calcula o seu valor e apresenta o resultado.*

```
> 5.3 + 7.2 * 0.1
6.02
> 2 < length [4,2,5,1]
True
> not True
False
```

- Um tipo é um nome que denota uma coleção de valores.
- Se da avaliação de uma expressão `e` resultar um valor do tipo `T`, então dizemos que a *expressão `e` tem tipo `T`*, e escrevemos

`e :: T`

Por exemplo,

```
> :type not True
not True :: Bool
```


Tipos

- Um tipo é um nome que denota uma coleção de valores. Por exemplo,
  - O tipo **`Bool`** contém os dois valores lógicos: **`True`** e **`False`**
  - O tipo **`Integer`** contém todos os números inteiros: **`…, -2, -1, 0, 1, 2, 3, …`**
- As funções só podem ser aplicadas a argumentos de tipo adequado. Por exemplo,

```
>  2 + True
error: …
```

Porque + deve ser aplicada a números e `True` não é um número.

- Se não houver concordância entre o tipo das funções e os seus argumentos, o programa é rejeitado pelo compilador.

Tipos

- Toda a expressão Haskell bem formada tem um tipo que é automaticamente calculado em tempo de compilação por um mecanismo chamado inferência de tipos.
- Por isso se diz que a linguagem Haskell é *"statically typed"*.
- Todos os erros de tipo são encontrados em tempo de compilação, o que torna os programas mais robustos.
- Os tipos permitem assim programar de forma mais produtiva, com menos erros.
- Num programa Haskell não é obrigatório escrever os tipos, o compilador infere-os, mas é boa prática escreve-los pois é uma forma de documentar o código.

Tipos básicos

Bool	Booleanos	`True, False`
Char	Caracteres	`'a', 'b', 'A', '3', '\n', …`
Int	Inteiros de tamanho limitado	`5, 7, 154243, -3452, …`
Integer	Inteiros de tamanho ilimitado	`26456118068672433368303400 43, …`
Float	Números de vírgula flutuante	`55.3, 23E5, 743.2e12, …`
Double	Números de vírgula flutuante de dupla precisão	`55.3, 23.5E5, …`
()	*Unit*	`()`

Tipos compostos

- **Tuplos** - sequências de tamanho fixo de elementos de diferentes tipos

```
(5, True) :: (Int, Bool)
(8, 3.5, 'b') :: (Int, Float, Char)
```

- **Listas** - sequências de tamanho variável de elementos de um mesmo tipo

```
[1,2,3,4,5] :: [Int]
[True, True, False] :: [Bool]
```

- **Funções** - mapeamento de valores de um tipo (o domínio da função) em valores de outro tipo (o contra-domínio da função)

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


Tuplos

Um tuplo é uma sequência de tamanho fixo de elementos que podem ser de diferentes tipos.

$(T_1, T_2, \ldots, T_n)$ é o tipo dos tuplos de tamanho n, cujo 1º elemento é de tipo T_1, o 2º elemento é de tipo T_2, …, e na posição n tem um elemento de tipo T_n.

Exemplos:

```
('C', 2, 'A') :: (Char, Int, Char)
(True, 1, False, 0) :: (Bool, Int, Bool, Int)
(3.5, ('a',True), 20) :: (Float, (Char,Bool), Int)
```

Listas

As listas são sequências de tamanho variável de elementos do mesmo tipo.

`[T]` é o tipo das listas de elementos do tipo `T`.

Exemplos:

```
[10,20,30] :: [Int]
[10, 20, 6, 19, 27, 30] :: [Int]
[True,True,False,True] :: [Bool]
[('a',True), ('b',False)] ::[(Char,Bool)]
[[3,2,1], [4,7,9,2], [5]] :: [[Int]]
```


Funções

Uma função é um mapeamento de valores de um tipo (o domínio da função) em valores de outro tipo (o contra-domínio da função)

T_1 `->` T_2 é o tipo das funções que *recebem* valores do tipo T_1 e *devolvem* valores do tipo T_2.

Exemplos:

```
even :: Int -> Bool
odd :: Int -> Bool
not :: Bool -> Bool
```


Funções com vários argumentos

Uma função com vários argumentos pode ser codificada de duas formas.

- Usando tuplos:

```
soma :: (Int,Int) -> Int
soma (x,y) = x + y
```

`soma` recebe um par de inteiros `(x,y)` e devolve o resultado inteiro `x+y`.

- Retornando funções como resultado:

```
add :: Int -> (Int -> Int)
add x y = x + y
```

`add` recebe um inteiro `x` e devolve uma função `(add x)`. Depois esta função recebe o inteiro `y` e devolve o resultado `x+y`.

Curried functions

```
soma :: (Int,Int) -> Int
add :: Int -> (Int -> Int)
```

`soma` e `add` produzem o mesmo resultado final, mas `soma` recebe os dois argumentos ao mesmo tempo, enquanto `add` recebe um argumento de cada vez.

- As funções que recebem os seus argumentos um de cada vez dizem-se *"curried"* em honra do matemático Haskell Curry que as estudou.
- Funções que recebem mais do que dois argumentos podem ser *curried* retornando funções aninhadas.

```
mult :: Int -> (Int -> (Int -> Int))
mult x y z = x * y * z
```

`mult` recebe um inteiro `x` e devolve uma função `(mult x)`, que por sua vez recebe o inteiro `y` e devolve a função `(mult x y)`, que finalmente recebe o inteiro `z` e devolve o resultado `x*y*z`.

Porque é que as funções *curried* são úteis?

- As funções *curried* são mais flexíveis porque é possível gerar novas funções, aplicando parcialmente uma função *curried*.

```
add 1 :: Int -> Int
```

- As funções Haskell são normalmente definidas na forma *curried*.

```
take :: Int -> [Int] -> [Int]

take 5 :: [Int] -> [Int]
```


Convenções

Para evitar o uso excessivo de parêntesis quando se usam funções *curried* são adotadas as seguintes convenções:

- A seta -> associa à direita

```
Int -> Int -> Int -> Int
```

Significa `Int -> (Int -> (Int -> Int))`

- A aplicação de funções associa à esquerda

```
mult x y z
```

Significa `((mult x) y) z`

Funções polimórficas

Há funções às quais é possível associar mais do que um tipo concreto. Por exemplo, a função `length` (que calcula o comprimento de uma lista) pode ser aplicada a quaisquer listas independentemente do tipo dos seus elementos.

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

Um função diz-se polimórfica se o seu tipo contém variáveis de tipo (representadas por letras minúsculas).

```
length :: [a] -> Int
```

Para qualquer tipo `a`, a função `length` recebe uma lista de valores do tipo `a` e devolve um inteiro.

Funções polimórficas

As variáveis de tipo podem ser instanciadas por diferentes tipos consoante as circunstâncias.

```
length :: [a] -> Int
```

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

> length [True,False]
2
```

`a = Int`

`a = Bool`

- As variáveis de tipo começam por letras minúsculas (normalmente usam-se as letras `a`, `b`, `c`, etc).
- O nome dos tipos concretos começa sempre por uma letra maiúscula (ex: `Int`, `Bool`, `Float`, etc)

Funções polimórficas

A maioria das funções da biblioteca `Prelude` são polimórficas.

```
head :: [a] -> a
tail :: [a] -> [a]
take :: Int -> [a] -> [a]
fst :: (a,b) -> a
snd :: (a,b) -> b
id :: a -> a
reverse :: [a] -> [a]
zip :: [a] -> [b] -> [(a,b)]
```


Sobrecarga (*overloading*) de funções

Considere os seguintes exemplos:

```
> 3 + 2
5
> 10.5 + 1.7
12.2
```

Qual será o tipo da soma (+) ?

Note que `a -> a -> a` é um tipo demasiado permissivo para a função (+), pois não é possível somar elementos de qualquer tipo

```
> 'a' + 'b'
:error …
```

O Haskell resolve o problema com restrições de classes

```
(+) :: Num a => a -> a -> a
```

Para qualquer tipo numérico a, a função `(+)` recebe dois valores do tipo a e devolve um valor do tipo a.

Sobrecarga (*overloading*) de funções

Uma função polimórfica diz-se sobrecarregada se o seu tipo contém uma ou mais restrições de classes.

```
(+) :: Num a => a -> a -> a
```

```
> 3 + 2
5
> 10.5 + 1.7
12.2
> 'a' + 'b'
error: …
```

`a = Int`

`a = Float`

`Char` não é um tipo numérico

Mais exemplos:

```
sum :: Num a => [a] -> a

(*) :: Num a => a -> a -> a

product :: Num a => [a] -> a
```


Class constraints

O Haskell tem muitas classes mas, por agora, apenas precisamos de ter a noção de que existem as seguintes

Num - a classe dos tipos numéricos (tipos sobre os quais estejam definidas operações como a soma e a multiplicação).

Eq - a classe dos tipos que têm o teste de igualdade definido.

Ord - a classe dos tipos que têm uma relação de ordem definida sobre os seus elementos.

```
(+) :: Num a => a -> a -> a

(==) :: Eq a => a -> a -> Bool

(<) :: Ord a => a -> a -> Bool
```

Mais tarde estudaremos em mais detalhe o mecanismo de classes do Haskell.

!!! Aviso !!!

Na versão actual do GHCi, se perguntarmos o tipo de certas funções sobre listas temos uma surpresa!!

```
> :type sum
sum :: (Foldable t, Num a) => t a -> a
```

Este é um tipo mais geral do que `sum :: Num a => [a] -> a` mas como as listas pertencem à classe `Foldable`, quando aplicamos `sum` a uma lista de números, este é o tipo efetivo da função `sum`.

Ao longo das aulas iremos sempre apresentar o tipo destas funções usando **listas** em vez da classe `Foldable`, para simplificar a apresentação.

Sempre que virem a classe `Foldable` num tipo das funções do Prelude, podem entender isso como sendo o tipo das listas.

Class constraints

Qual será o tipo das funções `elem` e `max` ?

```
> elem 3 [1,2,3,4,5]
True
> elem 7 [1,2,3,4,5]
False
> max 35 28
35
> max 5.6 10.7
10.7
```

```
elem :: Eq a => a -> [a] -> Bool
```

Porque se usa a função (==) na sua implementação.

```
max :: Ord a => a -> a -> a
```

Porque se usa a função (<) na sua implementação.

Mais alguns operadores do `Prelude`

Lógicos: `&&` (conjunção), `||` (disjunção), `not` (negação)

Numéricos: `+`, `-`, `*`, `/` (divisão de reais), `^` (exponenciação com inteiros), `div` (divisão inteira), `mod` (resto da divisão inteira), `abs` (módulo) `**` (exponenciações com reais), `log`, `sin`, `cos`, `tan`, …

Relacionais: `==` (igualdade), `/=` (desigualdade), `<`, `<=`, `>`, `>=`

Condicional: `if` … `then` … `else` …

`:: Bool` `:: a`

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 indentação do texto (isto é, o espaço entre a margem e o início do texto) 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 neste caso 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 (*pattern*) é uma variável, uma constante, ou um construtor aplicado a outros padrões. Ou seja, um padrão 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 fôrma" 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.

Nota: poderíamos definir, por exemplo, esta equação assim:

```
myAdd x y = Nothing
```


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.

Expressões `case`

O Haskell tem ainda uma forma construir expressões que permite fazer análise de casos sobre a estrutura dos valores de um tipo. Essas expressões têm a forma:

```
case expressão of
     padrão -> expressão
     ...
     padrão -> expressão
```

Exemplos:

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

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

Funções recursivas sobre listas

- Como definir a função que calcula o comprimento de uma lista ?
  - Sabemos calcular o comprimento da lista vazia: é zero.
  - Se soubermos o comprimento da cauda da lista, também sabemos calcular o comprimento da lista completa: basta somar-lhe mais um.
- Como as listas são construídas unicamente à custa da lista vazia e de acrescentar um elemento à cabeça da lista, a definição da função `length` é muito simples:

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```

Esta função é recursiva uma vez que se invoca a si própria.

- 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 função é aplicada à lista vazia.

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


Funções recursivas sobre listas

- **`sum`** calcula o somatório de uma lista de números.

```
sum :: Num a => [a] -> a
sum [] = 0
sum (x:xs) = x + sum xs
```

```
sum [1,2,3] = 1 + sum [2,3]
            = 1 + (2 + sum [3])
            = 1 + (2 + (3 + sum []))
            = 1 + 2 + 3 + 0
            = 6
```

- **`elem`** testa se um elemento pertence a uma lista.

```
elem :: Eq a => a -> [a] -> Bool
elem x [] = False
elem x (y:ys) | x == y    = True
              | otherwise = elem x ys
```

```
elem 2 [1,2,3] = elem 2 [2,3]
               = True
```

Passo 1: a 1ª equação que faz *match* é a 2ª, mas como a guarda é falsa, usa a 3ª equação.

Passo 2: usa a 2ª equação porque faz *match* e a guarda é verdadeira.

Funções recursivas sobre listas

- **`last`** dá o último elemento de uma lista não vazia.

Note como a equação **`last [x] = x`** tem que aparecer em 1º lugar.

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

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

O que aconteceria se trocássemos a ordem das equações?

Funções recursivas sobre listas

- **`init`** retira o último elemento de uma lista não vazia.

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

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

O que aconteceria se trocássemos a ordem das equações?

Funções recursivas sobre listas

- `(++)` faz a concatenação de duas listas.

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

Como a construção de listas é feita acrescentando elementos à esquerda da lista, vamos ter que definir a função fazendo a análise de casos sobre a lista da esquerda.

- Se a lista da esquerda for vazia

```
[] ++ l = l
```

- Se a lista da esquerda não for vazia

```
(x:xs) ++ l = x : (xs ++ l)
```

```
[1,2,3] ++ [4,5] = 1 : ([2,3] ++ [4,5])
                 = 1 : 2 : ([3] ++ [4,5])
                 = 1 : 2 : 3 : ([] ++ [4,5])
                 = 1 : 2 : 3 : [4,5]
                 = [1,2,3,4,5]
```

Haveria alguma diferença se trocássemos a ordem das equações?

Funções recursivas sobre listas

- `reverse` inverte uma lista.

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

Para acrescentar um elemento à direita da lista temos que usar `++[x]`

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


Funções recursivas sobre listas

- `(!!)` selecciona um elemento da lista numa dada posição.

```
(!!) :: [a] -> Int -> a
(x:xs) !! n
       | n == 0 = x
       | n > 0  = xs !! (n-1)
```

```
> [6,4,3,1,5,7]!!2
3
> [6]!!2
*** Exception: Non-exhaustive patterns
> [6,4,3,1,5,7]!!(-3)
*** Exception: Non-exhaustive patterns
```

```
[6,4,3,1,5,7]!!2 = [4,3,1,5,7]!!1
                 = [3,1,5,7]!!0
                 = 3
```

Porquê ?

Funções recursivas sobre listas

Exemplo: a função que soma uma lista de pares, componente a componente

```
somas :: [(Int,Int)] -> (Int,Int)
somas l = (sumFst l, sumSnd l)
```

O padrão `((x,y):t)` permite extrair as componentes do par que está na cabeça da lista.

```
sumFst :: [(Int,Int)] -> Int
sumFst [] = 0
sumFst ((x,y):t) = x + sumFst t
```

```
sumSnd :: [(Int,Int)] -> Int
sumSnd [] = 0
sumSnd ((x,y):t) = y + sumSnd t
```

- Esta função recorre às funções `sumFst` e `sumSnd`, como funções auxiliares, para fazer o cálculo dos resultados parciais.
- Há no entanto desperdício de trabalho nesta implementação, porque se está a percorrer a lista duas vezes sem necessidade.
- Numa só travessia podemos ir somando os valores das respectivas componentes.

Tupling *(calcular vários resultados numa só travessia da lista)*

- Numa só travessia podemos ir somando os valores das respectivas componentes, mantendo um par que vamos construindo.

```
somas :: [(Int,Int)] -> (Int,Int)
somas [] = (0,0)
somas ((x,y):t) = (x + fst (somas t), y + snd (somas t))
```

Note que `(soma t)` devolve um par. Daí o uso das funções `fst` e `snd`.

Pode parecer que `(somas t)` está a ser calculada duas vezes, mas isso não é verdade. `(somas t)` só é calculado uma vez, já que o valor dos identificadores é imutável.

- Podemos fazer uma declaração local para tornar o código mais fácil de ler

```
somas :: [(Int,Int)] -> (Int,Int)
somas [] = (0,0)
somas ((x,y):t) = (x + a, y + b)
   where (a,b) = somas t
```

Estamos aqui a usar o padrão `(a,b)` para extrair as componentes do par devolvido por `(somas t)`.

Tupling *(calcular vários resultados numa só travessia da lista)*

```
somas :: [(Int,Int)] -> (Int,Int)
somas [] = (0,0)
somas ((x,y):t) = (x + fst (somas t), y + snd (somas t))
```

```
somas [(7,8),(1,2)] = (7+ fst (somas [(1,2)]), 8+ snd (somas [(1,2)]))
                    = (7+ fst (1+ fst (somas []), 2+ fst (somas [])) ,
                       8+ snd (1+ fst (somas []), 2+ snd (somas [])) )
                    = (7+ fst (1+ fst (0,0), 2+ fst (0,0)) ,
                       8+ snd (1+ fst (0,0), 2+ snd (0,0)) )
                    = (7+ fst (1+0, 2+0) , 8+ snd (1+0, 2+0) )
                    = (7+1+0 , 8+2+0)
                    = (8, 10)
```


Tupling *(calcular vários resultados numa só travessia da lista)*

```
somas :: [(Int,Int)] -> (Int,Int)
somas [] = (0,0)
somas ((x,y):t) = (x+a, y+b)
   where (a,b) = somas t
```

```
somas [(7,8),(1,2)] = (7+ …, 8+ …) = (7+1+0, 8+2+0) = (8,10)
    somas [(1,2)] = (1+ …, 2+ …)   = (1+0, 2+0)
        somas [] = (0,0)
```


Funções recursivas sobre listas

- **`zip`** emparelha duas listas.

```
zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip _ _ = []
```

- **`unzip`** separa uma lista de pares em duas listas.

```
unzip :: [(a,b)] -> ([a],[b])
unzip [] = ([],[])
unzip ((x,y):t) = (x:e, y:d)
   where (e,d) = unzip t
```

```
unzip [(1,True),(2,False),(3,True)] = (1:… , True:…) = (1:2:3:[], True:False:True:[])
   unzip [(2,False),(3,True)] = (2:… , False:…)      = (2:3:[], False:True:[])
     unzip [(3,True)] = (3:… , True:…)               = (3:[], True:[])
       unzip [] = ([],[])
```

Funções recursivas sobre listas

- **take** dá os n primeiros elementos de uma lista.

```
take :: Int -> [a] -> [a]
take n _ | n <= 0 = []
take _ [] = []
take n (x:xs) = x : take (n-1) xs
```

```
take 2 [7,5,3] = 7 : take 1 [5,3]
               = 7 : 5 : take 0 [3]
               = 7 : 5 : []
               = [7,5]
```

```
take 2 [7] = 7 : take 1 []
           = 7 : []
           = [7]
```


Funções recursivas sobre listas

- **drop** retira os n primeiros elementos de uma lista.

```
drop :: Int -> [a] -> [a]
drop n xs | n <= 0  = xs
drop _ [] = []
drop n (_:xs) = drop (n-1) xs
```

```
drop 2 [7,5,3] = drop 1 [5,3]
               = drop 0 [3]
               = [3]
```

```
drop 2 [7] = drop 1 []
           = []
```


Tuppling

- **splitAt** parte uma lista em duas da seguinte forma

```
splitAt :: Int -> [a] -> ([a],[a])
splitAt n l = (take n l, drop n l)
```

Esta função recorre às funções `take` e `drop` como funções auxiliares. Há no entanto algum desperdício de trabalho nesta implementação, porque se está a percorrer a lista até à posição `n` duas vezes sem necessidade.

Podemos definir a função assim

```
splitAt :: Int -> [a] -> ([a],[a])
splitAt n l | n <= 0  = ([],l)
splitAt _ [] = ([],[])
splitAt n (x:xs) = (x:l1, l2)
     where (l1,l2) = splitAt (n-1) xs
```

```
splitAt 2 [7,8,9,0,1] = (7:… , …)                    = (7:8:[] , [9,0,1])
   splitAt 1 [8,9,0,1] = (8:… , …)      = (8:[] , [9,0,1])
       splitAt 0 [9,0,1] = ([],[9,0,1])
```


Lazy evaluation

- O Haskell faz o cálculo do valor de uma expressão usando as equações que definem as funções como regras de cálculo.
- Cada passo do processo de cálculo costuma chamar-se de redução.
- Cada redução resulta de substituir a instância do lado esquerdo da equação pelo respectivo lado direito.
- A estratégia de redução usada para o cálculo das expressões é uma característica essencial de uma linguagem funcional.

Exemplo: considere as seguintes funções

```
dobro x = x + x
snd (x,y) = y
```

Como é que a expressão `dobro (snd (3,7))` é calculada?

Há duas possibilidades:

```
dobro (snd (3,7)) = dobro 7 = 7 + 7 = 14
```

ou

```
dobro (snd (3,7)) = snd (3,7) + snd (3,7) = 7 + 7 = 14
```

Lazy evaluation

- O Haskell usa como estratégia de redução a *lazy evaluation* (também chamada de *call-by-name*).
- A *lazy evaluation* caracteriza-se por aplicar as funções sem fazer o cálculo prévio dos seus argumentos.
- A sequência de redução que o Haskell faz no cálculo da expressão `dobro (snd (3,7))` é

```
dobro (snd (3,7)) = snd (3,7) + snd (3,7) = 7 + 7 = 14
```

- Com a *lazy evaluation* os argumentos das funções só são calculados se o seu valor fôr mesmo necessário.

```
snd (sqrt (20^3+ (45/23)^10), 1) = 1
```

- A *lazy evaluation* faz do Haskell uma linguagem não estrita. Isto é, uma função aplicada a um valor indefinido pode ter em Haskell um valor bem definido.

```
snd (5 `div` 0, 1) = 1
```

- A *lazy evaluation* também vai permitir ao Haskell lidar com estruturas de dados infinitas.

```
take 7 [5,10..] = [5,10,15,20,25,30,35]
```


Algoritmos de ordenação

- A ordenação de um conjunto de valores é um problema muito frequente e muito útil na organização de informação.
- Para resolver o problema de ordenação de uma lista existem muitos algoritmos. Vamos estudar três desses algoritmos:
  - *Insertion sort*
  - *Quick sort*
  - *Merge sort*
- Vamos apresentar esses algoritmos para ordenar uma lista por ordem crescente.

Insertion sort

Este algoritmo apoia-se numa função auxiliar que insere um elemento numa lista já ordenada.

```
insert :: (Ord a) => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) | x<=y = x:y:ys
                | otherwise = y : insert x ys
```

```
insert 7 [2,5,9]
     = 2 : insert 7 [5,9]
     = 2 : 5 : insert 7 [9]
     = 2:5:7:9:[]
     = [2,5,7,9]
```

A função de ordenação da lista define-se por casos:

```
isort :: (Ord a) => [a] -> [a]
isort [] = []
isort (x:xs) = insert x (isort xs)
```

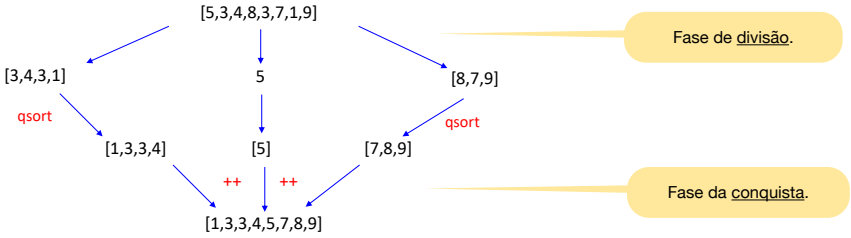
Se a lista não é vazia, insere o elemento da cabeça da lista na cauda previamente ordenada pelo mesmo método.

```
isort [4,7,2] = insert 4 (isort [7,2])
              = insert 4 (insert 7 (isort [2]))
              = insert 4 (insert 7 (insert 2 (isort [])))
              = insert 4 (insert 7 (insert 2 []))
              = insert 4 (insert 7 [2]) = …
              = insert 4 [2,7] = … = [2,4,7]
```


Quick sort

Este algoritmo segue uma estratégia chamada de "divisão e conquista".

- Quando a lista não é vazia, selecciona-se a cabeça da lista e parte-se a cauda em duas listas:
  - uma lista com os elementos que são mais pequenos do que a cabeça,
  - e outra lista com os restantes elementos (isto é, os que são maiores ou iguais à cabeça)
- Depois ordenam-se estas listas mais pequenas pelo mesmo método.
- Finalmente concatenam-se as listas ordenadas e a cabeça, de forma a que a lista final fique ordenada.



Quick sort

```
qsort :: (Ord a) => [a] -> [a]
qsort [] = []
qsort (x:xs) = (qsort l1) ++ [x] ++ (qsort l2)
   where (l1,l2) = parte x xs
```

A função `parte` pode ser feita usando a técnica de *tuppling*

```
parte :: (Ord a) => a -> [a] -> ([a],[a])
parte _ [] = ([],[])
parte x (y:ys) | y < x = (y:as, bs)
               | otherwise = (as, y:bs)
    where (as,bs) = parte x ys
```

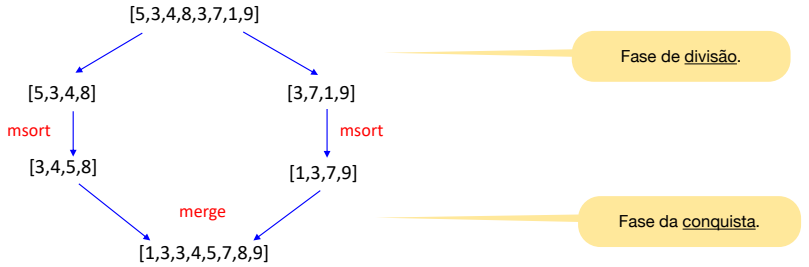
```
qsort [4,7,2] = (qsort …) ++ [4] ++ (qsort …) = (qsort [2]) ++ [4] ++ (qsort [7])
                                              = … = [2] ++ [4] ++ [7] = [2,4,7]

   parte 4 [7,2] = (…, 7:…)                   = (2:[], 7:[])
        parte 4 [2] = (2:…, …)                = (2:[],[])
             parte 4 [] = ([],[])
```


Merge sort

Este algoritmo segue uma estratégia de "divisão e conquista".

- Quando a lista tem mais do que um elemento, parte-se a lista em duas listas de tamanho aproximadamente igual (podem diferir em uma unidade).
- Depois ordenam-se estas listas mais pequenas pelo mesmo método.
- Finalmente faz-se a fusão das duas listas ordenadas de forma a que a lista final fique ordenada.



Merge sort

Começemos pela função `merge` que faz a fusão de duas listas ordenadas.

```
merge :: (Ord a) => [a] -> [a] -> [a]
merge [] l = l
merge l [] = l
merge (x:xs) (y:ys) | x < y = x : merge xs (y:ys)
                    | otherwise = y : merge (x:xs) ys
```

A função de ordenação pode definir-se assim:

```
msort :: (Ord a) => [a] -> [a]
msort [] = []
msort [x] = [x]
msort l = merge (msort l1) (msort l1)
   where n = (length l) `div` 2
         (l1,l2) = splitAt n l
```

```
msort [4,7,2] = merge (msort [4]) (msort [7,2]) = … = merge [4] [2,7]
                                                = 2 : merge [4] [7]
                                                = 2 : 4 : merge [] [7]
                                                = 2 : 4 : [7] = [2,4,7]
 porque  splitAt 1 [4,7,2] = ([4], [7,2])
```


Merge sort

Podemos definir a função `msort` de outro modo:

`nome@padrão` é uma forma de fazer uma definição local ao nível de um argumento de uma função.

```
merge :: (Ord a) => [a] -> [a] -> [a]
merge [] l = l
merge l [] = l
merge a@(x:xs) b@(y:ys) | x < y = x : merge xs b
                        | otherwise = y : merge a ys
```

`split` parte a lista numa só travessia. A lista está ser partida de maneira diferente, mas isso não tem interferência no algoritmo.

```
split :: [a] -> ([a],[a])
split [] = ([],[])
split [x] = ([x],[])
split (x:y:t) = (x:l1, y:l2)
   where (l1,l2) = split t
```

```
msort :: (Ord a) => [a] -> [a]
msort [] = []
msort [x] = [x]
msort l = merge (msort l1) (msort l1)
   where (l1,l2) = split l
```

Funções com parâmetro de acumulação

- A ideia que está na base destas funções é que elas vão ter um parâmetro extra (o acumulador) onde a resposta vai sendo construída e gravada à medida que a recursão progride.
- O acumulador vai sendo actualizado e passado como parâmetro nas sucessivas chamadas da função.
- Uma vez que o acumulador vai guardando a resposta da função, o seu tipo deve ser igual ao tipo do resultado da função.

Exemplo: A função que inverte uma lista.

A função `inverte` chama uma função auxiliar `inverteAc` com um parâmetro de acumulação e inicializa o acumulador.

O acumulador é inicialmente a lista vazia.

```
inverte :: [a] -> [a]
inverte l = inverteAc l []
  where inverteAc [] ac = ac
        inverteAc (x:xs) ac = inverteAc xs (x:ac)
```

Quando a lista é vazia o acumulador tem a solução completa.

A chamada recursiva é feita actualizando o acumulador.

Funções com parâmetro de acumulação

```
inverte :: [a] -> [a]
inverte l = inverteAc l []
  where inverteAc [] ac = ac
        inverteAc (x:xs) ac = inverteAc xs (x:ac)
```

```
inverte [1,2,3] = inverteAc [1,2,3] []
                = inverteAc [2,3] [1]
                = inverteAc [3] [2,1]
                = inverteAc [] [3,2,1]
                = [3,2,1]
```

A solução está a ser construída no acumulador.

Esta versão é bastante mais eficiente que a função `reverse` anteriormente definida (porque usa o `++` que tem que atravessar a primeira lista).

```
reverse :: [a] -> [a]
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]
```


Funções com parâmetro de acumulação

Podemos sistematizar as seguintes regras para definir funções usando esta técnica:

1. Colocar o acumulador como um parâmetro extra.
2. O acumulador deve ser do mesmo tipo que o do resultado da função.
3. Devolver o acumulador no acaso de paragem da função.
4. Actualizar o acumulador na chamada recursiva da função.
5. A função principal (sem acumulador) chama a função com parâmetro de acumulação, inicializando o acumulador.

Exemplo: O somatório de uma lista de números.

```
somatorio :: Num a => [a] -> a
somatorio l = sumAc l 0
  where sumAc :: Num a => [a] -> a -> a
        sumAc [] n = n
        sumAc (x:xs) n = sumAc xs (x+n)
```

```
somatorio [1,2,3]
        = sumAc [1,2,3] 0
        = sumAc [2,3] (1+0)
        = sumAc [3] (2+1+0)
        = sumAc [] (3+2+1+0)
        = 6
```


Funções com parâmetro de acumulação

Exemplo: O máximo de uma lista não vazia.

```
maximo :: Ord a => [a] -> a
maximo (x:xs) = maxAc xs x
  where maxAc :: Ord a => [a] -> a -> a
        maxAc [] n = n
        maxAc (x:xs) n = if x > n then maxAc xs x
                                  else maxAc xs n
```

```
maximo [2,7,3,9,4] = maxAc [7,3,9,4] 2
                   = maxAc [3,9,4] 7
                   = maxAc [9,4] 7
                   = maxAc [4] 9
                   = maxAc [] 9
                   = 9
```

Funções com parâmetro de acumulação

Exemplo: A função factorial.

```
factorial :: Integer -> Integer
factorial n = factAc n 1
  where factAc :: Integer -> Integer -> Integer
        factAc 0 x = x
        factAc n x | n>0 = factAc (n-1) (n*x)
```

```
factorial 5 = factAc 5 1
            = factAc 4 (5*1)
            = factAc 3 (4*5*1)
            = factAc 2 (3*4*5*1)
            = factAc 1 (2*3*4*5*1)
            = factAc 0 (1*2*3*4*5*1)
            = 120
```


Funções com parâmetro de acumulação

Exemplo: A função `stringToInt :: String -> Int` que converte uma string (representando um número inteiro positivo) num valor inteiro.

```
stringToInt "5247" = 5247
```

```
import Data.Char

stringToInt :: String -> Int
stringToInt (x:xs) = aux xs (digitToInt x)
  where aux :: String -> Int -> Int
        aux (h:t) ac = aux t (ac*10 + (digitToInt h))
        aux [] ac = ac
```

```
stringToInt "5247" = aux "247" 5
                   = aux "47" (50+2)
                   = aux "7" (520+4)
                   = aux "" (5240+7)
                   = 5247
```


Listas por compreensão

Na matemática é costume definir conjuntos por compreensão à custa de outros conjuntos.

$\{ 2x \mid x \in \{10,3,7,2\} \}$ — O conjunto $\{20,6,14,4\}$.

$\{ n \mid n \in \{4,-5,8,20,-7,1\} \wedge 0 \leq n \leq 10 \}$ — O conjunto $\{4,8,1\}$.

Em Haskell podem definir-se listas por compreensão, de modo semelhante, construindo novas listas à custa de outras listas.

```
[ 2*x | x <- [10,3,7,2] ]
```
A lista `[20,6,14,4]`.

```
[ n | n <- [4,-5,8,20,-7,1], 0<=n, n<=10 ]
```
A lista `[4,8,1]`.

Listas por compreensão

A expessão `x <- [1,2,3,4,5]` é chamada de gerador da lista.

A expessão `10 <= x^2` é uma guarda que restringe os valores produzidos pelo gerador que a precede.

```
> [ x^2 | x <- [1,2,3,4,5], 10 <= x^2 ]
[16,25]
```

As listas por compreensão podem ter vários geradores e várias guardas.

```
> [ (x,y) | x <- [1,2,3], y <- [4,6] ]
[(1,4),(1,6),(2,4),(2,6),(3,4),(3,6)]
```

Mudar a ordem dos geradores muda a ordem dos elementos na lista final.

```
> [ (x,y) | y <- [4,5], x <- [1,2,3] ]
[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
```

Um gerador pode depender de variáveis introduzidas por geradores anteriores.

```
> [ (x,y) | x <- [1..3], y <- [x..3] ]
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

Listas por compreensão

Pode-se usar a notação `..` para representar uma enumeração com o passo indicado pelos dois primeiros elementos. Caso não se indique o segundo elemento, o passo é um.

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

```
> [1,10..100]
[1,10,19,28,37,46,55,64,73,82,91,100]
```

```
> [20,15..(-7)]
[20,15,10,5,0,-5]
```

```
> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
```


Listas infinitas

É possível também definir listas infinitas.

```
> [1..]
[1,2,3,4,5,6,7,8,9,10,11,…
```

```
> [0,10..]
[0,10,20,30,40,50,60,70,80,90,100,110,120,130,…
```

```
> [ x^3 | x <- [0..], even x ]
[0,8,64,216,512,1000,…
```

```
> take 10 [3,3..]
[3,3,3,3,3,3,3,3,3,3]
```

```
> zip "Haskell" [0..]
[('H',0),('a',1),('s',2),('k',3),('e',4),('l',5),('l',6)]
```


Funções e listas por compreensão

Podem-se definir funções usando listas por compreensão.

Exemplo: A função de ordenação de listas *quick sort.*

```
qsort :: (Ord a) => [a] -> [a]
qsort [] = []
qsort (x:xs) = (qsort [y | y<-xs, y<x]) ++[x]++ (qsort [y | y<-xs, y>=x])
```

Esta versão do *quick sort* faz duas travessias da lista para fazer a sua partição e, por isso, é pior do que a versão anterior com a função auxiliar `parte`.

Funções e listas por compreensão

Exemplo: Usando a função `zip` e listas por compreensão, podemos definir a função que calcula a lista de posições de um dado valor numa lista.

```
posicoes :: Eq a => a -> [a] -> [Int]
posicoes x l = [ i | (y,i) <- zip l [0..], x == y]
```

O lado esquerdo do gerador da lista é um padrão.

A *lazy evaluation* do Haskell faz com que não seja problemático usar uma lista infinita como argumento da função `zip`.

```
> posicoes 3 [4,5,3,4,5,6,3,5,3,1]
[2,6,8]
```

Funções e listas por compreensão

Exemplo: Calcular os divisores de um número positivo.

```
divisores :: Integer -> [Integer]
divisores n = [ x | x <- [1..n], n `mod` x == 0]
```

Testar se um número é primo.

```
primo :: Integer -> Bool
primo n = divisores n == [1,n]
```

```
> primo 5
True
> primo 1
False
```

Lista de números primos até um dado n.

```
primosAte :: Integer -> [Integer]
primosAte n = [ x | x <- [1..n], primo x]
```

Lista infinita de números primos.

```
primos :: [Integer]
primos = [ x | x <- [2..], primo x]
```


Crivo de Eratóstenes

Um algoritmo mais eficiente para encontrar números primos, é o Crivo de Eratóstenes (assim chamado em honra ao matemático grego que o inventou), que permite obter todos os números primos até um determinados valor n. A ideia é a seguinte:

- Começa-se com a lista [2..n].
- Guarda-se o primeiro elemento da lista (pois é um número primo) e removem-se da cauda da lista todos os múltiplos desse primeiro elemento.
- Continua-se a aplicar o passo anterior à restante lista, até que a lista de esgote.

```
crivo []     = []
crivo (x:xs) = x : crivo [ n | n <- xs , n `mod` x /= 0 ]
```

```
primosAte n = crivo [2..n]
```

Lista infinita de números primos.

```
primos = crivo [2..]
```


Factorização em primos

O Teorema Fundamental da Aritmética (enunciado pela primeira vez por Euclides) diz que qualquer número inteiro (maior do que 1) pode ser decomposto num produto de números primos. Esta decomposição é única a menos de uma permutação.

Exemplo: Com o auxílio da lista de números primos, podemos definir uma função que dado um número (maior do que 1), calcula a lista dos seus factores primos.

```
factoriza :: Integer -> [Integer]
factoriza n = aux n primos
  where aux 1 _ = []
        aux n (x:xs)
            | n `mod` x /= 0 = aux n xs
            | otherwise      = x : aux (n `div` x) (x:xs)
```

```
> factoriza 94753
[19,4987]
> factoriza 9475312
[2,2,2,2,7,11,7691]
```


Funções de ordem superior

Em Haskell, as funções são entidades de primeira ordem. Ou seja,

- As funções podem receber outras funções como argumento.

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)
```

Exemplos:

```
dobro :: Int -> Int
dobro x = x + x
```

```
quadruplo :: Int -> Int
quadruplo x = twice dobro x
```

```
quadruplo 5 = twice dobro 5
            = dobro (dobro 5)
            = (dobro 5) + (dobro 5)
            = (5+5) + (5+5)
            = 10 + 10
            = 20
```

```
retira2 :: [a] -> [a]
retira2 l = twice tail l
```

```
retira2 [4,5,7,0,9] = twice tail [4,5,7,0,9]
                    = tail (tail [4,5,7,0,9])
                    = tail [5,7,0,9]
                    = [7,0,9]
```

Funções de ordem superior

- As funções podem devolver outras funções como resultado.

```
mult :: Int -> Int -> Int
mult x y = x * y
```

O tipo é igual a `Int -> (Int -> Int)`, porque `->` é associativo à direita

Exemplos:

```
triplo :: Int -> Int
triplo = mult 3
```

`triplo` tem o mesmo tipo que `mult 3`

```
triplo 5 = mult 3 5
         = 3 * 5
         = 15
```

`mult 3 5 = (mult 3) 5`, porque a aplicação é associativa à esquerda

```
twice (mult 2) 5 = (mult 2) ((mult 2) 5) = mult 2 (mult 2 5)
                 = 2 * (mult 2 5)
                 = 2 * (2 * 5)
                 = 20
```


map

Consideremos as seguintes funções:

```
triplos :: [Int] -> [Int]
triplos [] = []
triplos (x:xs) = 3*x : triplos xs
```

Estas funções fazem coisas distintas entre si, mas **a forma como operam é semelhante**: aplicam uma transformação a cada elemento da lista de entrada.

```
maiusculas :: String -> String
maiusculas [] = []
maiusculas (x:xs) = toUpper x : maiusculas xs
```

Dizemos que estas funções têm um **padrão de computação** comum, e apenas diferem na função que é aplicada a cada elemento da lista.

```
somapares :: [(Float,Float)] -> [Float]
somapares [] = []
somapares ((a,b):xs) = a+b : somapares xs
```

A função **`map`** do `Prelude` sintetiza este padrão de computação, abstraindo em relação à função que é aplicada aos elementos da lista.

map

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = (f x) : (map f xs)
```

`map` é uma função de ordem superior que recebe a função `f` que é aplicada ao longo da lista.

Exemplos:

```
triplos :: [Int] -> [Int]
triplos l = map (3*) l
```

```
triplos [1,2] = map (3*) [1,2]
              = 3*1 : map (3*) [2]
              = 3*1 : 3*2 : map (3*) []
              = 3*1 : 3*2 : []
              = 3:6:[] = [3,6]
```

```
maiusculas :: String -> String
maiusculas xs = map toUpper xs
```

```
somapares :: [(Float,Float)] -> [Float]
somapares l = map aux l
     where aux (a,b) = a+b
```

Usando listas por compreensão, poderíamos definir a função `map` assim:

```
map f l = [ f x | x <- l ]
```


filter

Consideremos as seguintes funções:

```
pares :: [Int] -> [Int]
pares [] = []
pares (x:xs) = if even x
               then x : pares xs
               else pares xs
```

Estas funções fazem coisas distintas entre si, mas **a forma como operam é semelhante**: selecionam da lista de entrada os elementos que verificam uma dada condição.

```
positivos :: [Double] -> [Double]
positivos [] = []
positivos (x:xs)
         | x > 0     = x : positivos xs
         | otherwise = positivos xs
```

Estas funções têm um **padrão de computação** comum, e apenas diferem na condição com que cada elemento da lista é testado.

A função **`filter`** do `Prelude` sintetiza este padrão de computação, abstraindo em relação à condição com que os elementos da lista são testados.

filter

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs)
     | p x       = x : filter p xs
     | otherwise = filter p xs
```

`filter` é uma função de ordem superior que recebe a condição `p` (um predicado) com que cada elemento da lista é testado.

Exemplos:

```
pares :: [Int] -> [Int]
pares l = filter even l
```

```
pares [1,2,3,4] = filter even [1,2,3,4]
                = filter even [2,3,4]
                = 2 : filter even [3,4]
                = 2 : filter even [4]
                = 2 : 4 : filter even []
                = 2:4:[] = [2,4]
```

```
positivos :: [Double] -> [Double]
positivos xs = filter (>0) xs
```

Usando listas por compreensão, poderíamos definir a função `filter` assim:

```
filter p l = [  x | x <- l, p x ]
```


Funções anónimas

Em Haskell é possível definir funções sem lhes dar nome, através expressões lambda.

Por exemplo, `\x -> x+x`

É uma **função anónima** que recebe um número `x` e devolve como resultado `x+x`.

```
> (\x -> x+x) 5
10
```

Uma expressão lambda tem a seguinte forma (a notação é inspirada no *λ-calculus*):

\padrão ... padrão -> expressão

Exemplos:

```
> (\x y -> x+y) 3 8
11
```

```
> (\(x1,y1) (x2,y2) -> (x1+x2,y1+y2)) (3,2) (7,9)
(10,11)
```

```
> (\(x:xs) -> xs) [1,2,3]
[2,3]
```

```
> (\(x:xs) y -> y:xs) [1,2,3] 9
[9,2,3]
```


Funções anónimas

As expressões lambda são úteis para evitar declarações de pequenas funções auxiliares.

Exemplo: Em vez de

```
trocapares :: [(a,b)] -> [(b,a)]
trocapares l = map troca l
   where troca (x,y) = (y,x)
```

pode-se escrever

```
trocapares l = map (\(x,y)->(y,x)) l
```

Exemplo:

```
multiplosDe :: Int -> [Int] -> [Int]
multiplosDe n xs = filter (\x -> mod x n == 0) xs
```


Funções anónimas

As expressões lambda podem ser usadas na definição de funções. Por exemplo:

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

```
soma1 = \x y -> x + y
```

`soma`, `soma1`, `soma2` e `soma3` são funções equivalentes

```
soma2 = \x -> (\y -> x + y)
```

```
soma3 x = \y -> x + y
```

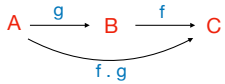
Os operadores infixos aplicados apenas a um argumento (a que se dá o nome de secções), são uma forma abreviada de escrever funções anónimas.

Exemplos:

```
(+y)  ≡  \x -> x+y
(x+)  ≡  \y -> x+y
(*3)  ≡  \x -> x*3
```

Funções de ordem superior

- `(.)` composição de funções




```
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) f g x = f (g x)
```

Exemplo:

```
ultimo :: [a] -> a
ultimo = head . reverse
```

```
ultimo [1,2,3] = (head . reverse) [1,2,3]
               = head (reverse [1,2,3])
               = head [3,2,1]
               = 3
```

- `flip` troca a ordem dos argumentos de uma função binária.

```
flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x
```

```
> (^) 3 2
9
> flip (^) 3 2
8
```

Exemplo:

```
mytake :: [a] -> Int -> [a]
mytake = flip take
```

```
mytake [1..10] 3 = flip take [1..10] 3
                 = take 3 [1..10]
                 = [1,2,3]
```


Funções de ordem superior

- `curry` transforma uma função que recebe como argumento um par, numa função equivalente que recebe um argumento de cada vez.

```
curry :: ((a,b) -> c) -> a -> b -> c
curry f x y = f (x,y)
```

- `uncurry` transforma uma função que recebe dois argumentos (um de cada vez), numa função equivalente que recebe um par.

```
uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (x,y) = f x y
```

Exemplo:

```
quocientes :: [(Int,Int)] -> [Int]
quocientes l = map (\(x,y) -> div x y) l
```

Ou, em alternativa,

```
quocientes l = map (uncurry div) l
```

```
> quocientes [(10,3), (20,4)]
[3,5]
```


Funções de ordem superior

- `zipWith` constrói uma lista cujos elementos são calculados por uma função que é aplicada a argumentos que vêm de duas listas.

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) =  f x y : zipWith f xs ys
zipWith _ _ _ = []
```

```
> zipWith div [10,20..50] [1..]
[10,10,10,10,10]
```

```
> zipWith (^) [1..5] [2,2..]
[1,4,9,16,25]
```

```
> map (uncurry (^)) (zip [1..5] [2,2..])
[1,4,9,16,25]
```


Funções de ordem superior

- `takeWhile` recebe uma condição e uma lista e retorna o segmento inicial da lista cujos elementos satisfazem a condição dada.

```
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p [] =  []
takeWhile p (x:xs) | p x       = x : takeWhile p xs
                   | otherwise = []
```

```
> takeWhile (>3) [5,7,1,8,2]
[5,7]
```

- `dropWhile` recebe uma condição e uma lista e retorna a lista sem o segmento inicial de elementos que satisfazem a condição dada.

```
dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile p [] =  []
dropWhile p (x:xs) | p x       = dropWhile p xs
                   | otherwise = x:xs
```

```
> dropWhile (>3) [5,7,1,8,2]
[1,8,2]
```

Funções de ordem superior

- **span** é uma função do Prelude que calcula simultamente o resultado das funções takeWhile e dropWhile. Ou seja, `span p l == (takeWhile p l, dropWhile p l)`

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

Exemplo: A função **lines** (do Prelude) que parte uma string numa lista de linhas.

```
> lines "  \nabds\tbfsas\n26egd\n\n3673qw"
["  ","abds\tbfsas","26egd","","3673qw"]
```

```
lines :: String -> [String]
lines [] = []
lines s = let (l, r) = span (/='\n') s
          in case r of
               []  -> [l]
               x:xs -> l : lines xs
```


Funções de ordem superior

- **break** é uma função do Prelude equivalente à função span invocada com a condição negada.

```
break :: (a -> Bool) -> [a] -> ([a],[a])
break p l = span (not . p) l
```

```
> break (>10) [3,4,5,30,8,12,9]
([3,4,5],[30,8,12,9])
```

Exemplo: A função **words** (do Prelude) que parte uma string numa lista de palavras.

```
> words "  \nabds\tbfsas\n26egd\n\n3673qw"
[“abds","bfsas","26egd","3673qw"]
```

```
words :: String -> [String]
words [] = []
words s = let l = dropWhile isSpace s
              (a,b) = break isSpace l
          in a : words b
```


foldr *(right fold)*

Considere as seguintes funções:

Estas funções fazem coisas distintas entre si, mas **a forma como operam é semelhante**: aplicam um operador binário à cabeça da lista e ao resultado de aplicar a função à cauda da lista, e quando a lista é vazia devolvem um determinado valor.

```
and [] =  True
and (b:bs) = b && (and bs)
```

```
product [] = 1
product (x:xs) = x * (product xs)
```

Estas funções têm um **padrão de computação** comum. Apenas diferem no operador binário que é usado e no valor a devolver quando a lista é vazia.

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

```
concat [] = []
concat (l:ls) = l ++ (concat ls)
```

A função **foldr** do Prelude sintetiza este padrão de computação, abstraindo em relação ao operador binário que é usado e ao resultado a devolver quando a lista é vazia.

foldr *(right fold)*

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

foldr é uma função de ordem superior que recebe o operador **f** que é usado para construir o resultado, e o valor **z** a devolver quando a lista é vazia.

Exemplos:

```
and :: [Bool] -> Bool
and l = foldr (&&) True l
```

```
product :: Num a => [a] -> a
product xs = foldr (*) 1 xs
```

```
sum :: Num a => [a] -> a
sum l = foldr (+) 0 l
```

```
concat :: [[a]] -> [a]
concat l = foldr (++) [] l
```

```
sum [1,2,3] = foldr (+) 0 [1,2,3]
      = 1 + (foldr (+) 0 [2,3])
      = 1 + (2 + (foldr (+) 0 [3]))
      = 1 + (2 + (3 + (foldr (+) 0 [])))
      = 1 + (2 + (3 + 0)))
      = 6
```

Note que `foldr f z [x1,…,xn] == f x1 (… (f xn z)…)`
`== x1 `f` (… (xn `f` z)…)`
Ou seja, aplica `f` **associando à direita**.

foldr

Podemos olhar para a expressão `(foldr f z l)` como a substituição de cada `(:)` da lista por `f` , e de `[]` por `z`.

```
foldr f z (x1:x2:…:xn:[]) == x1 `f` (x2 `f` (… `f` (xn `f` z) …))
```

O resultado vai sendo construído a partir do lado direito da lista.

Exemplos:

```
foldr (+) 0 [1,2,3] == 1 + (2 + (3 + 0)))
```

```
foldr (*) 1 [1,2,3] == 1 * (2 * (3 * 1)))
```

```
foldr (&&) True [False,True,False] == False && (True && (False && True)))
```

```
foldr (++) [] [“abc”,“zzzz”,“bb”] == “abc” ++ (“zzzz” ++ (“bb” ++ []))
```


foldr

Muitas funções (mais do que à primeira vista poderia parecer) podem ser definidas usando o `foldr`.

Exemplo:

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

`x` representa a cabeça da lista e `r` o resultado da chamada recursiva sobre a cauda.

Pode ser definida assim:

```
reverse l = foldr (\x r -> r++[x]) [] l
```

Exemplo:

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```

`h` representa a cabeça da lista e `r` o resultado da chamada recursiva sobre a cauda.

Pode ser definida assim:

```
length = foldr (\h r -> 1+r) 0
```


foldl *(left fold)*

Existe no `Prelude` uma outra função, **`foldl`**, que vai construindo o resultado pelo lado esquerdo da lista.

```
foldl f z [x1,x2,…,xn] == (… ((z `f` x1) `f` x2) …) `f` xn
```

Exemplo:

```
foldl (+) 0 [1,2,3] == ((0 + 1) + 2) + 3
```

A função **`foldl`** sintetiza um padrão de computação que corresponde a trabalhar com um acumulador. O `foldl` recebe como argumentos a função que combina o acumulador com a cabeça da lista, e o valor inicial do acumulador.

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs
```

`z` é o acumulador. `f` é usado para combinar o acumulador com a cabeça da lista.

`(f z x)` é o novo valor do acumulador.

foldl *(left fold)*

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs
```

`z` é o acumulador. `f` é usado para combinar o acumulador com a cabeça da lista. `(f z x)` é o novo valor do acumulador.

Exemplo: Vejamos a relação entre função somatório implementada com um acumulador

```
sumAc [] ac = ac
sumAc (x:xs) ac = sumAc xs (ac+x)
```

```
sumAc [1,2,3] 0 = sumAc [2,3] (0+1)
                = sumAc [3] ((0+1)+2)
                = sumAc [] (((0+1)+2)+3)
                = ((0+1)+2)+3
                = 6
```

e o somatório implementado com um `foldl`

```
foldl (+) 0 [1,2,3] = foldl (+) (0+1) [2,3]
                    = foldl (+) ((0+1)+2) [3]
                    = foldl (+) (((0+1)+2)+3) []
                    = ((0+1)+2)+3
                    = 6
```

foldl

Muitas funções (mais do que à primeira vista poderia parecer) podem ser definidas usando o `foldl`.

Exemplo:

```
inverte :: [a] -> [a]
inverte l = inverteAc l []
  where inverteAc [] ac = ac
        inverteAc (x:xs) ac = inverteAc xs (x:ac)
```

Pode ser definida assim:

```
inverte l = foldl (\ac x -> x:ac)) [] l
```

`ac` representa o valor acumulado e `x` a cabeça da lista. `[]` é o valor inicial do acumulador.

Ou assim:

```
inverte l = foldl (flip (:)) [] l
```

Exemplo: A função `stringToInt :: String -> Int` definida anteriormente, com um parâmetro de acumulação, pode ser definida de forma equivalente assim:

```
stringToInt l = foldl (\ac x -> 10*ac + digitToInt x) 0 l
```


foldr vs foldl

Note que as expressões **`(foldr f z xs)`** e **`(foldl f z xs)`** só darão o mesmo resultado se a função **`f`** for comutativa e associativa, caso contrário dão resultados distintos.

Exemplo:

```
foldr (-) 8 [4,7,3,5] = 4 – (7 – (3 – (5 - 8)))
                      = 3
```

```
foldl (-) 8 [4,7,3,5] = (((8 - 4) – 7) – 3) – 5
                      = -11
```


Tipos algébricos

A construção de tipos algébricos dá à linguagem Haskell um enorme poder expressivo, pois permite a implementação de tipos enumerados, co-produtos (união disjunta de tipos), e tipos indutivos (recursivos).

O tipo das listas é um exemplo de um tipo indutivo (recursivo) :

```
data [a] = []
         | (:) a [a]
```

Uma lista,

- ou é vazia,
- ou tem um elemento e a uma sub-estrutura que é também uma lista.

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

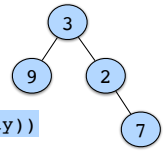
A noção de árvore binária expande este conceito.

Uma árvore binária,

- ou é vazia,
- ou tem um elemento e a duas sub-estruturas que são também árvores.

```
data BTree a = Empty
             | Node a (BTree a) (BTree a)
```

```
Node 3 (Node 9 Empty Empty) (Node 2 Empty (Node 7 Empty Empty))
```



Árvores binárias

As árvores binárias são estruturas de dados muito úteis para organizar a informação.

```
data BTree a = Empty
             | Node a (BTree a) (BTree a)
  deriving (Show)
```

Os construtores da árvores são:

```
Empty :: BTree a

Node :: a -> (BTree a) -> (BTree a) -> (BTree a)
```

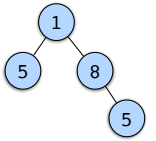
`Empty` representa a árvore vazia.

`Node` recebe um elemento e duas árvores, e constrói a árvore com esse elemento na raiz, uma árvore do lado esquerdo e outra do lado direito.

```
arv1 = Node 5 Empty Empty
```

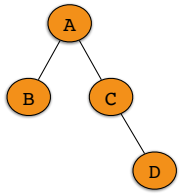

```
arv2 = Node 8 Empty arv1
```


```
arv3 = Node 1 arv1 arv2
```



Árvores binárias

Terminologia



- O nodo A é a **raiz** da árvore.
- Os nodos B e C são **filhos** (ou **descendentes**) de A.
- O nodo C é **pai** de D.
- B e C são **folhas** da árvore.
- O **caminho** (***path***) de um nodo é a sequência de nodos da raiz até esse nodo. Por exemplo, A,C,D é o caminho para o modo D.
- A **altura** da árvore é o comprimento do caminho mais longo. Esta árvore tem altura 3.

Árvores binárias

As funções definidas sobre tipos de dados recursivos, são geralmente funções recursivas, com padrões de recursividade semelhantes aos dos tipos de dados.

Exemplo: Calcular o número de nodos que tem uma árvore.

```
conta :: BTree a -> Int
conta Empty = 0
conta (Node x e d) = 1 + conta e + conta d
```

Exemplo: Somar todos de nodos de uma árvore de números .

```
sumBT :: Num a =>  BTree a -> a
sumBT Empty = 0
sumBT (Node x e d) = x + sumBT e + sumBT d
```

```
> sumBT (Node 2 Empty (Node 7 Empty Empty))
        = 2 + (sumBT Empty) + sumBT (Node 7 Empty Empty)
        = 2 + 0 + (7 + sumBT Empty + sumBT Empty)
        = 2 + 0 + (7 + 0 + 0)
        = 9
```


Árvores binárias

Exemplo: Calcular a altura de uma árvore.

```
altura :: BTree a -> Int
altura Empty = 0
altura (Node _ e d) = 1 + max (altura e) (altura d)
```

Exemplos: As funções map e zip para árvores binárias.

```
mapBT :: (a -> b) -> BTree a -> BTree b
mapBT f Empty = Empty
mapBT f (Node x e d) = Node (f x) (mapBT f e) (mapBT f d)
```

```
zipBT :: BTree a -> BTree b -> BTree (a,b)
zipBT (Node x1 e1 d1) (Node x2 e2 d2) =
                          Node (x1,x2) (zipBT e1 e2) (zipBT d1 d2)
zipBT _ _ = Empty
```


Travessias de árvores binárias

Uma árvore pode ser percorrida de várias formas. As principais estratégias são:

Travessia preorder: visitar a raiz, depois a árvore esquerda e a seguir a árvore direita.

```
preorder :: BTree a -> [a]
preorder Empty = []
preorder (Node x e d) = [x] ++ (preorder e) ++ (preorder d)
```

Travessia inorder: visitar árvore esquerda, depois a raiz e a seguir a árvore direita.

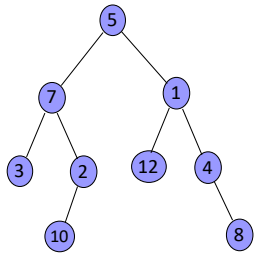
```
inorder :: BTree a -> [a]
inorder Empty = []
inorder (Node x e d) = (inorder e) ++ [x] ++ (inorder d)
```

Travessia postorder: visitar árvore esquerda, depois árvore direita, e a seguir a raiz..

```
postorder :: BTree a -> [a]
postorder Empty = []
postorder (Node x e d) = (postorder e) ++ (postorder d) ++ [x]
```

Travessias de árvores binárias

```
arv = (Node 5 (Node 7 (Node 3 Empty Empty)
                      (Node 2 (Node 10 Empty Empty) Empty)
              )
              (Node 1 (Node 12 Empty Empty)
                      (Node 4 Empty (Node 8 Empty Empty))
              )
      )
```




```
preorder arv  = [5,7,3,2,10,1,12,4,8]

inorder arv   = [3,7,10,2,5,12,1,4,8]

postorder arv = [3,10,2,7,12,8,4,1,5]
```

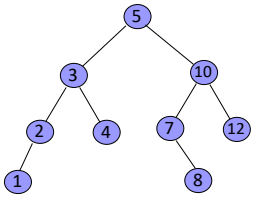

Árvores binárias de procura

Uma árvore binária em que o valor de cada nodo é maior do que os nodos à sua esquerda, e menor do que os nodos à sua direita diz-se uma árvore binária de procura (ou de pesquisa)

Uma **árvore binária de procura** é uma árvore binária que verifica as seguinte condição:

- a raiz da árvore é maior do que todos os elementos que estão na sub-árvore esquerda;
- a raiz da árvore é menor do que todos os elementos que estão na sub-árvore direita;
- ambas as sub-árvores são árvores binárias de procura.

Exemplo: Esta é uma árvore binária de procura de procura



Árvores binárias de procura

Exemplo: Testar se um elemento pertence a uma árvore binária de procura.

```
elemBT :: Ord a => a -> BTree a -> Bool
elemBT x Empty = False
elemBT x (Node y e d)
          | x < y  = elemBT x e
          | x > y  = elemBT x d
          | x == y = True
```

Exemplo: Inserir um elemento numa árvores binária de procura.

```
insertBT :: Ord a => a -> BTree a -> BTree a
insertBT x Empty = Node x Empty Empty
insertBT x (Node y e d)
          | x < y  = Node y (insertBT x e) d
          | x > y  = Node y e (insertBT x d)
          | x == y = Node y e d
```

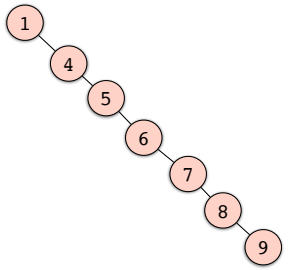

Árvores binárias de procura

O formato de uma árvore depende da ordem pela qual os elementos vão sendo inseridos.

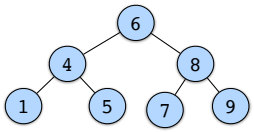
Exemplo: Considere a seguinte função que converte uma lista numa árvore, inserindo os elementos pela a ordem em que estes aparecem na lista.

```
listToBT :: Ord a => [a] -> BTree a
listToBT l = foldl (flip insertBT) Empty l
```

```
listToBT [1,4,5,6,7,8,9] =
```




```
listToBT [6,4,1,8,9,5,7] =
```

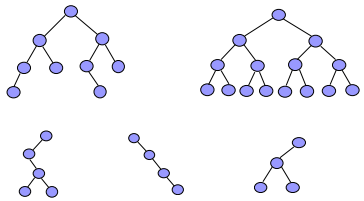


Árvores balanceadas

Uma árvore binária diz-se **balanceada** (ou **equilibrada**) se é vazia, ou se verifica as seguintes condições:

- as alturas das sub-árvores esquerda e direita diferem no máximo em uma unidade;
- ambas as sub-árvores são balanceadas.

Exemplos:



Balanceadas.

Desbalanceadas.

Exercício: Defina uma função que testa se uma árvore é balanceada.

Árvores binárias de procura

As árvores binárias de procura possibilitam pesquisas potencialmente mais eficientes do que as listas.

Exemplo:

Pesquisa numa lista não ordenada.

```
lookup :: Eq a => a -> [(a,b)] -> Maybe b
lookup x [] = Nothing
lookup x ((y,z):t) | x == y = Just z
                   | x /= y = lookup x t
```

Pesquisa numa lista ordenada.

```
lookupSL :: Ord a => a -> [(a,b)] -> Maybe b
lookupSL x [] = Nothing
lookupSL x ((y,z):t) | x < y  = Nothing
                     | x == y = Just z
                     | x > y  = lookupSL x t
```

O número de comparações de chaves é **no máximo igual ao comprimento da lista.**

Pesquisa numa árvore binária de procura.

```
lookupBT :: Ord a => a -> BTree (a,b) -> Maybe b
lookupBT x Empty = Nothing
lookupBT x (Node (y,z) e d) | x < y  = lookupBT x e
                            | x > y  = lookupBT x d
                            | x == y = Just z
```

O número de comparações de chaves é **no máximo igual à altura da árvore**.

Árvores binárias de procura

A pesquisa em árvores binárias de procura são especialmente mais eficientes se as árvores forem balanceadas.

Uma forma de balancear uma árvore de procura consiste em: primeiro gerar uma lista ordenada com os seus elementos e depois, a partir dessa lista, gerar a árvore.

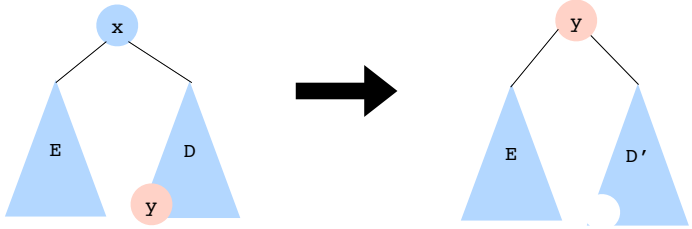
```
balance :: BTree a -> BTree a
balance t = constroi (inorder t)

constroi :: [a] -> BTree a
constroi [] = Empty
constroi l = let n = length l
                 (l1, x:l2) = splitAt (n `div` 2) l
             in Node x (constroi l1) (constroi l2)
```

Exercício: Defina uma versão mais eficiente desta função que não esteja a calcular sempre o comprimento da lista. (Sugestão: trabalhe com um par que tem a lista e o seu comprimento.)

Árvores binárias de procura

A remoção do elemento que está na raiz de uma árvore de procura pode ser feita indo buscar o menor elemento da sub-árvore direita (ou, em alternativa, o maior elemento da sub-árvore esquerda) para tomar o seu lugar.



Exercício: Com base nesta ideia, defina uma função que remove um elemento uma árvore de procura. Comece por definir uma função que devolve um par com o mínimo de uma árvore não vazia e a árvore sem o mínimo.

Árvores irregulares *(rose trees)*

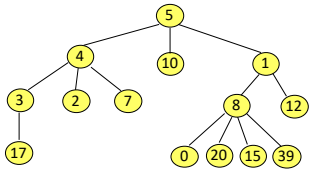
Nas **árvores irregulares** cada nodo pode ter um número variável de descendentes. O seguinte tipo de dados é uma implementação de árvores irregulares, não vazias.

```
data RTree a = R a [RTree a]
   deriving (Show)
```

O único construtor da árvore é `R :: a -> [RTree a] -> RTree a`

R recebe o elemento que fica na raiz da árvore e a lista das sub-árvores com que vai construir a árvore.

```
R 5 [ R 4 [ R 3 [R 17 []], R 2 [], R 7 []],
      R 10 [],
      R 1 [ R 8 [ R 0 [], R 20 [], R 15 [],  R 39 [] ],
            R 12 [] ]
    ]
```



Árvores irregulares *(rose trees)*

Como é de esperar, as funções definidas sobre rose trees seguem um padrão de recursividade compatível com sua definição indutiva.

Exemplo: Contar os nodos de uma árvore.

```
contaRT :: RTree a -> Int
contaRT (R x l) = 1 + sum (map contaRT l)
```

Exemplo: Calcular a altura de uma árvore.

```
alturaRT :: RTree a -> Int
alturaRT (R x []) = 1
alturaRT (R x l) = 1 + maximum (map alturaRT l)
```


Árvores irregulares *(rose trees)*

Exemplo: Testar se um elemento pertence a uma árvore.

```
pertenceRT :: Eq a => a -> RTree a -> Bool
pertenceRT x (R y l) = x==y || or (map (pertenceRT x) l)
```

`(pertenceRT x) :: RTree a -> Bool`

Exemplo: Fazer uma travessia *preorder* uma árvore.

```
preorderRT :: RTree a -> [a]
preorderRT (R x l) = x : concat (map preorderRT l)
```

Exercício: Defina uma função que converte uma árvore binária numa *rose tree.*

Outras árvores

Leaf trees: Árvores binárias em que a informação está apenas nas folhas da árvore. Os nós intermédios não têm informação.

```
data LTree a = Tip a
             | Fork (LTree a) (LTree a)
```

Full trees: Árvores binárias que têm informação nos nós intermédios e nas folhas. A informação guardada nos nós e nas folhas pode ser de tipo diferente.

```
data FTree a b = Leaf b
               | No a (FTree a b) (FTree a b)
```

Overloading

- Em Haskell é possível usar o mesmo identificador para funções computacionalmente distintas. A isto chama-se sobrecarga (*overloading*) de funções.
- Ao nível do sistema de tipos a sobrecarga de funções é tratada introduzindo o conceito de classe e **tipos qualificados**.

Exemplo:

```
(+) :: Num a => a -> a -> a
```

```
> 3 + 2
5
> 10.5 + 1.7
12.2
> 'a' + 'b'
error: …
```

a = Int que pertence à classe Num

a = Float que pertence à classe Num

Char não pertence à classe Num

Classes & instâncias

- As **classes** são uma forma de classificar tipos quanto às funcionalidades que lhe estão associadas.
  - Uma classe estabelece um conjunto de assinaturas de funções (i.e., seu nome e tipo).
  - Os tipos que são declarados como instâncias dessa classe têm que ter essas funções definidas.

Exemplo: A declaração (simplificada) da classe `Num`

```
class Num a where
    (+) :: a -> a -> a
    (*) :: a -> a -> a
```

exige que todo o tipo `a` da classe `Num` tenha que ter as funções `(+)` e `(*)` definidas

Para declarar `Int` e `Float` como pertencendo à classe `Num` têm que se fazer as seguintes declarações de instância:

```
instance Num Int where
    (+) = primPlusInt
    (*) = primMulInt
```

```
instance Num Float where
    (+) = primPlusFloat
    (*) = primMulFloat
```


Classes & instâncias

```
instance Num Int where
    (+) = primPlusInt
    (*) = primMulInt
```

```
instance Num Float where
    (+) = primPlusFloat
    (*) = primMulFloat
```

Neste caso as funções `primPlusInt`, `primMulInt`, `primPlusFloat` e `primMulFloat` são funções primitivas da linguagem.

```
> 3 + 2
5
> 10.5 + 1.7
12.2
> 7 * 3
21
> 3.4 * 2.0
6.8
```

3 `primPlusInt` 2

10.5 `primPlusFloat` 1.7

7 `primMultInt` 3

3.4 `primMultFloat` 2.0

Tipos principais

O **tipo principal** de uma expressão é o tipo mais geral que lhe é possível associar, de forma a que todas as possíveis instâncias desse tipo constituam ainda tipos válidos para a expressão.

- Toda a expressão válida tem um tipo principal único.
- O Haskell infere sempre o tipo principal de uma expressão.

Exemplo: Podemos definir uma classe `FigFechada`

```
class FigFechada a where
    area :: a -> Float
    perimetro :: a -> Float
```

e definir a função `areaTotal` que calcula o total das áreas das figuras que estão numa lista

```
areaTotal l = sum (map area l)
```

```
> :type areaTotal
areaTotal :: (FigFechada a) => [a] -> Float
```

A classe Eq

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    -- Minimal complete definition: (==) or (/=)
    x == y = not (x /= y)
    x /= y = not (x == y)
```

Esta classe estabelece as funções `(==)` e `(/=)` e, para além disso, fornece também definições por omissão para estas funções *(default methods)*.

- Caso a definição de uma função seja omitida numa declaração de instância, o sistema assume a definição feita na classe.
- Se existir uma nova definição do método na declaração de instância, será essa definição a ser usada.

A classe Eq

Exemplo: Considere a seguinte definição do tipo dos números naturais

```
data Nat = Zero
         | Suc Nat
```

Um valor do tipo `Nat` ou é `Zero`, ou é `Suc n`, em que `n` é do tipo `Nat`

Os valores do tipo `Nat` são portanto, `Zero, Suc Zero, Suc (Suc Zero), …`

O tipo `Nat` pode ser declarado como instância da classe `Eq` assim:

```
instance Eq Nat where
    (Suc n) == (Suc m) = n == m
    Zero == Zero       = True
    _ == _             = False
```

A função `(/=)` fica definida por omissão.

Esta declaração de instância, por testar a **igualdade literal (estrutural)** entre dois valores do tipo `Nat`, poderia ser derivada automaticamente fazendo

```
data Nat = Zero | Suc Nat
   deriving (Eq)
```


A classe Eq

Nem sempre a igualdade estrutural (que testa se dois valores são iguais quando resultam do mesmo construtor aplicado a argumentos também iguais) é o que precisamos.

Exemplo: Considere o seguinte tipo para representar horas em dois formatos distintos.

```
data Time = AM Int Int | PM Int Int | Total Int Int
```

Queremos, por exemplo, que `(PM 3 30)` e (Total 15 30) sejam iguais, pois representam a mesma hora do dia.

Exercício: Defina uma função que converte para minutos um valor `Time` e, com base nela, declare `Time` como instância da classe `Eq`.

Instâncias com restrições

Exemplo: Considere o tipo das árvores binárias.

```
data BTree a = Empty | Node a (BTree a) (BTree a)
```

Só poderemos declarar `(BTree a)` como instância da classe `Eq` se o tipo `a` for também uma instância da classe `Eq`. Este tipo de restrição pode ser colocado na declaração de instância, fazendo:

Igualdade sobre valores do tipo `(BTree a)`

```
instance (Eq a) => Eq (BTree a) where
    Empty == Empty = True
    (Node x1 e1 d1) == (Node x2 e2 d2) = (x1==x2) && (e1==e2)&& (d1==d2)
    _ == _  = False
```

Igualdade sobre valores do tipo `a`

Esta declaração de instância poderia ser derivada automaticamente fazendo:

```
data BTree a = Empty | Node a (BTree a) (BTree a)
   deriving (Eq)
```

Herança

O sistema de classes do Haskell tem um mecanismo de **herança**. Por exemplo, podemos definir a classe **Ord** como uma ***extensão*** da classe **Eq**.

```
class (Eq a) => Ord a where
    compare               :: a -> a -> Ordering
    (<), (<=), (>=), (>)  :: a -> a -> Bool
    max, min              :: a -> a -> a
    …
```

- A classe `Ord` **herda** todas as funções da classe `Eq` e, além disso, estabelece um conjunto de operações de comparação e as funções máximo e mínimo.
- Diz-se que `Eq` é uma **superclasse** de `Ord`, ou que `Ord` é uma **subclasse** de `Eq`.
- Todo o tipo que é instância de `Ord` tem necessáriamente de ser instância de `Eq`.

A classe Ord

```
data Ordering = LT | EQ | GT
```

```
class (Eq a) => Ord a where
    compare               :: a -> a -> Ordering
    (<), (<=), (>=), (>)  :: a -> a -> Bool
    max, min              :: a -> a -> a
    -- Minimal complete definition: (<=) or compare
    -- using compare can be more efficient for complex types
    compare x y | x == y    = EQ
                | x <= y    = LT
                | otherwise = GT
    x <= y                  = compare x y /= GT
    x <  y                  = compare x y == LT
    x >= y                  = compare x y /= LT
    x >  y                  = compare x y == GT
    max x y   | x <= y      = y
              | otherwise   = x
    min x y   | x <= y      = x
              | otherwise   = y
```

Para declarar um tipo como instância da classe **Ord**, basta definir a função **(<=)** ou a função **compare**

A classe Ord

Exemplo: Declarar `Nat` como instância da classe `Ord`

```
data Nat = Zero | Suc Nat
   deriving (Eq)
```

pode ser feito assim:

```
instance Ord Nat where
    compare (Suc _ ) Zero    = GT
    compare Zero (Suc _)     = LT
    compare Zero Zero        = EQ
    compare (Suc x) (Suc y) = compare x y
```

```
> Suc Zero <= Suc (Suc Zero)
True
```

A função (<=) fica definida por omissão.

Ou, em alternativa, assim:

```
instance Ord Nat where
    Zero <= _          = True
    (Suc x) <= (Suc y) = x <= y
    (Suc _) <=  Zero   = False
```


A classe Ord

Exemplo: Declarar `Time` como instância da classe `Ord`

```
data Time = AM Int Int
          | PM Int Int
          | Total Int Int
```

```
totalmin :: Time -> Int
totalmin (AM h m) = h*60 + m
totalmin (PM h m) = (12+h)*60 + m
totalmin (Total h m) = h*60 + m
```

```
instance Eq Time where
  t1 == t2 = (totalmin t1) == (totalmin t2)
```

É necessário que **Time** seja da classe **Eq**.

pode agora ser feito assim:

```
instance Ord Time where
  t1 <= t2 = (totalmin t1) <= (totalmin t2)
```

Exemplo de uma função que usa o operador `(<)` definido por omissão:

```
select :: Time -> [(Time,String)] -> [(Time,String)]
select t l = filter ((t<) . fst) l
```

Este é o **(<=)** para o tipo **Int**. Note que **Int** é instância da classe **Ord**.

A classe Show

A classe **Show** estabelece métodos para converter um valor de um tipo qualquer numa string.

O interpretador Haskell usa a função **show** para apresentar o resultado dos seu cálculos.

```
class Show a where
    show      :: a -> String
    showsPrec :: Int -> a -> ShowS
    showList  :: [a] -> ShowS
    -- Minimal complete definition: show or showsPrec
    show x          = showsPrec 0 x ""
    showsPrec _ x s = show x ++ s
    showList []     = showString "[]"
    showList (x:xs) = showChar '[' . shows x . showl xs
        where showl []     = showChar ']'
              showl (x:xs) = showChar ',' . shows x . showl xs
```

Basta definir a função **show**. O restante fica definido por omissão.

```
type ShowS = String -> String
```

```
shows :: Show a => a -> ShowS
shows = showsPrec 0
```

A função showsPrec usa uma string como acumulador. É mais eficiente.

A classe Show

Exemplo: Declarar `Nat` como instância da classe `Show` de forma a que os naturais sejam apresentados do modo usual

```
natToInt :: Nat -> Int
natToInt Zero = 0
natToInt (Suc n) = 1 + (natToInt n)
```

```
> Suc (Suc Zero)
2
```

```
instance Show Nat where
    show n = show (natToInt n)
```

Este é o **show** para o tipo **Int**. Note que **Int** é instância da classe **Show**.

Instâncias da classe `Show` podem ser derivadas automaticamente. Neste caso, o método **show** produz uma string com o mesmo aspecto do valor que lhe é passado como argumento.

Ficaria assim:

```
data Nat = Zero | Suc Nat
   deriving (Eq,Show)
```

```
> Suc (Suc Zero)
Suc (Suc Zero)
```


A classe Show

Exemplo: Declarar `Time` como instância da classe `Show`

```
instance Show Time where
   show (AM h m) = (show h) ++ ":" ++ (show m) ++ " am"
   show (PM h m) = (show h) ++ ":" ++ (show m) ++ " pm"
   show (Total h m) = (show h) ++ "h" ++ (show m) ++ "m"
```

```
> AM 4 30
4:30 am
> Total 17 45
17h45m
```

A função **show** é usada para apresentar o valos valores no ghci.

A função **showList**, definida por omissão, é usada pelo ghci para apresentar a lista.

```
> [(PM 43 20), (AM 2 15), (Total 17 30)]
[43:20 pm,2:15 am,17h30m]
```


A classe Num

A classe **Num** está no topo de uma hierarquia de classes numéricas desenhada para controlar as operações que devem estar definidas sobre os diferentes tipos de números. Os tipos **Int**, **Integer**, **Float** e **Double**, são instâncias desta classe.

Note que **Num** é subclasse das classes **Eq** e **Show**.

```
class (Eq a, Show a) => Num a where
    (+), (-), (*)  :: a -> a -> a
    negate         :: a -> a
    abs, signum    :: a -> a
    fromInteger    :: Integer -> a
    -- Minimal complete definition: All, except negate or (-)
    x - y           = x + negate y
    negate x        = 0 - x
```

A função `fromInteger` converte um `Integer` num valor do tipo Num a => a.

```
> :type 35
35 :: Num a => a
> 35 + 5.7
40.7
```

35 é na realidade **(fromInteger 35)**

A classe Enum

A classe **Enum** estabelece um conjunto de operações que permitem **sequências aritméticas**.

```
class Enum a where
    succ, pred           :: a -> a
    toEnum               :: Int -> a
    fromEnum             :: a -> Int
    enumFrom             :: a -> [a]             -- [n..]
    enumFromThen         :: a -> a -> [a]        -- [n,m..]
    enumFromTo           :: a -> a -> [a]        -- [n..m]
    enumFromThenTo       :: a -> a -> a -> [a]   -- [n,n'..m]
    -- Minimal complete definition: toEnum, fromEnum
    succ                  = toEnum . (1+)       . fromEnum
    pred                  = toEnum . subtract 1 . fromEnum
    enumFrom x            = map toEnum [ fromEnum x ..]
    enumFromThen x y      = map toEnum [ fromEnum x, fromEnum y ..]
    enumFromTo x y        = map toEnum [ fromEnum x .. fromEnum y ]
    enumFromThenTo x y z  = map toEnum [ fromEnum x, fromEnum y .. fromEnum z ]
```

Entre as instâncias desta classe contam-se os tipos: `Int, Integer, Float, Double, Char, …`

```
> [2,2.5 .. 4]
[2.0,2.5,3.0,3.5,4.0]
```

```
> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
```


A classe Enum

Exemplo: `Time` como instância da classe `Enum`

```
instance Enum Time where
    toEnum n = let (h,m) = divMod n 60
               in  Total h m
    fromEnum = totalmin
```

```
> [(AM 1 0), (AM 2 30) .. (PM 1 0)]
[1h0m,2h30m,4h0m,5h30m,7h0m,8h30m,10h0m,11h30m,13h0m]
> [(PM 2 25) .. (Total 14 30)]
[14h25m,14h26m,14h27m,14h28m,14h29m,14h30m]
```

É possível derivar automaticamente instâncias da classe `Enum`, apenas em **tipos enumerados**.

Exemplo:

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

```
> [Amarelo .. Azul]
[Amarelo,Verde,Vermelho,Azul]
```


A classe Read

A classe **Read** estabelece funções que são usadas na conversão de uma string num valor do tipo de dados (instância de `Read`) quando isso é possível.

```
class Read a where
    readsPrec :: Int -> ReadS a
    readList  :: ReadS [a]
    -- Minimal complete definition: readsPrec
    readList   = ...
```

```
read :: Read a => String -> a
read s = case [x | (x,t) <- reads s, ("","") <- lex t] of
              [x] -> x
              []  -> error "Prelude.read: no parse"
              _   -> error "Prelude.read: ambiguous parse"
```

lex é um analizador léxico do `Prelude`.

```
type ReadS a = String -> [(a,String)]
```

```
reads :: Read a => ReadS a
reads = readsPrec 0
```

Podemos definir instâncias da classe **Read** que permitam fazer o *parser* do texto de acordo com uma determinada sintaxe, mas isso não é tópico de estudo nesta disciplina.

A classe Read

Instâncias da classe `Read` podem ser **derivadas automaticamente**. Neste caso, a função **read**, recebendo uma string que obedeça às regras sintáticas de Haskell, produz o valor do tipo correspondente.

Exemplos:

```
data Time = AM Int Int
          | PM Int Int
          | Total Int Int
   deriving (Read)
```

```
data Nat = Zero | Suc Nat
   deriving (Eq,Read)
```

Quase todos os tipos pré-definidos pertencem à classe **Read**

```
> read "AM 8 30" :: Time
8:30 am
> read "(Total 17 15)" :: Time
17h15m
> read "Suc (Suc Zero)" :: Nat
2
> read "5+4" :: Int
*** Exception: Prelude.read: no parse
```

```
> read "[2,3,6,7]" :: [Int]
[2,3,6,7]
> read "[(AM 2 3), Total 5 6]" :: [Time]
[2:3 am,5h6m]
> read "[Zero, Suc Zero]" :: [Nat]
[0,1]
```

Classes de construtores de tipos

Quando se faz uma declaração de um tipo algébrico, introduzem-se construtores de valores e construtores de tipos. Por exemplo,

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

`Maybe` é um constructor de tipo.

`Nothing` e `Just` são construtores de valores do tipo `Maybe a`

Em Haskell é possível definir classes de construtores de tipos.

Exemplo:

```
class Functor f where
    fmap :: (a -> b) -> (f a -> f b)
```

`f` é um construtor de tipo

Podemos declarar os constructores de tipos lista, BTree e Maybe como instância da classe Functor.

```
instance Functor [] where
    fmap = map
```

```
instance Functor BTree where
    fmap = mapBT
```

```
instance Functor Maybe where
    fmap f Nothing  = Nothing
    fmap f (Just x) = Just (f x)
```


Monads

O conceito de **mónade** é usado para sintetizar a ideia de computação. Uma computação é algo que se passa dentro de uma **"caixa negra"** e da qual conseguimos apenas ver os resultados.

Monad é uma classe de construtores de tipos do Haskell.

`return` corresponde a uma computação nula

`(>>=)` compõe computações aproveitando o valor devolvido pela primeira para o cálculo da segunda.

`(>>)` compõe computações ignorando o valor devolvido pela primeira no cálculo da segunda.

```
class Monad m where
    return :: a -> m a
    (>>=)  :: m a -> (a -> m b) -> m b   -- "bind"
    (>>)   :: m a -> m b -> m b          -- "sequence"
    fail   :: String -> m a
    -- Minimal complete definition: (>>=), return
    p >> q  = p >>= \ _ -> q
    fail s  = error s
```

`t :: m a` significa que `t` é uma computação que retorna um valor do tipo `a`. Ou seja, `t` é um valor do tipo `a` com um efeito adicional captado por `m`. Este efeito pode ser, por exemplo, uma acção de IO.

Input / Output

Como conciliar o princípio de "computação por cálculo" com o IO ?

Exemplo: Qual será o tipo de uma função `lerChar` que lê um caracter do teclado?

```
lerChar :: Char
```

?

Se assim fosse, `lerChar` seria uma constante do tipo `Char` !!!

- As funções do Haskell são funções matemáticas puras.
- Ler do teclado é um efeito lateral.
- Os programas interactivos têm efeitos laterais.
- As funções interactivas podem ser escritas em Haskell usando o construtor de tipos **`IO`** , para distiguir expressões puras de acções impuras que podem envolver efeitos laterais.
- **`(IO a)`** é o tipo das acções de input/output que retornam um valor do tipo **`a`**.
- **`IO`** é instância da classe `Monad`.
- A função que lê do teclado um caracter é `getChar :: IO Char`

`getChar` é um valor do tipo `Char` que resulta de uma acção de input/output.

Algumas funções IO do Prelude

- Para ler do *standard input* (por omissão, o teclado):

```
getChar  :: IO Char
getLine  :: IO String
```

lê um caracter;
lê uma string.

- Para escrever no *standard ouput* (por omissão, o ecrã):

```
putChar  :: Char -> IO ()
putStr   :: String -> IO ()
putStrLn :: String -> IO ()
print    :: Show a => a -> IO ()
```

escreve um caracter;
escreve uma string;
escreve uma string e muda de linha;
equivalente a (putStrLn . show)

- Para lidar com ficheiros de texto:

```
writeFile  :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
readFile   :: FilePath -> IO String
```

escreve uma string no ficheiro;
acrescenta no final do ficheiro;
lê o conteúdo do ficheiro para uma string.

```
type FilePath = String
```

é o nome do ficheiro (pode incluir a *path* no *file system*).

Monad IO

O monade IO agrega os tipos de todas as computações onde existem acções de input/output.

- `return :: a -> IO a` não faz nenhuma acção de IO. Apenas faz a conversão de tipo.
- `(>>=) :: IO a -> (a -> IO b) -> IO b` compõe duas acções de IO podendo utilizar o valor devolvido pela primeira para o cálculo da segunda.
- `(>>) :: IO a -> IO b -> IO b` compõe duas acções de IO de forma independente.

Exemplos: já definidos no Prelude

```
putStr :: String -> IO ()
putStr [] = return ()
putStr (x:xs) = (putChar x) >> (putStr xs)
```

```
getLine :: IO String
getLine = getChar >>= (\x-> if x=='\n'
                            then return []
                            else getLine >>= (\xs-> return (x:xs)) )
```


Notação "do"

O Haskell fornece uma construção sintática (**do**) para escrever de forma simplificada cadeias de operações monádicas.

Exemplos: Podemos escrever

```
do e1
   e2
```
ou
```
do { e1; e2 }
```
em vez de `e1 >> e2`

```
do x <- e1
   e2
```
em vez de `e1 >>= (\x -> e2)`

```
do x1 <- e1
   x2 <- e2
   e3
```
em vez de `e1 >>= (\x1-> e2 >>= (\x2-> e3))`

```
do e                                  ≡  e
do e1; e2;...; en                     ≡  e1 >> do e2;...; en
do x <- e1; e2;...; en                ≡  e1 >>= \ x -> do e2;...; en
do let declarações; e2;...; en        ≡  let declarações in do e2;...; en
```


Notação "do"

Exemplos:

```
putStr :: String -> IO ()
putStr [] = return ()
putStr (x:xs) = do putChar x
                   putStr xs
```

```
getLine :: IO String
getLine = do x <- getChar
             if x=='\n'
             then return []
             else do xs <- getLine
                     return (x:xs)
```

Exemplo: Combinando "do" e "let"

```
test :: IO ()
test = do putStr "Escreva uma frase: "
          l <- getLine
          let a = map toUpper l
              b = map toLower l
          putStrLn ("Maiúsculas: " ++ a)
          putStr ("Minúsculas: " ++ b)
```

```
> test
Escreva uma frase: aEIou
Maiúsculas: AEIOU
Minúsculas: aeiou
```


Notação "do"

Exemplo: Defina a função `dialogo` que escreve no ecrã uma pergunta e recolhe a resposta dada.

```
dialogo :: String -> IO String
dialogo s = do putStr s
               r <- getLine
               return r
```

```
dialogo' :: String -> IO String
dialogo' s = (putStr s) >> (getLine >>= (\r -> return r))
```

Exemplo: Defina a função `questionario` que recebe uma lista de questões e devolve a lista com as respostas dadas interactivamente .

```
questionario :: [String] -> IO [String]
questionario [] = return []
questionario (q:qs) = do r <- dialogo q
                         rs <- questionario qs
                         return (r:rs)
```

Input / Output

Exemplo: Cálculo das raizes de um polinómio de 2º grau.

```
roots :: (Float,Float,Float) -> Maybe (Float,Float)
roots (a,b,c)
    | d >= 0 = Just ((-b + (sqrt d))/(2*a), (-b - (sqrt d))/(2*a))
    | d < 0  = Nothing
  where d = b^2 - 4*a*c
```

Camada interactiva:

```
calcRoots :: IO ()
calcRoots =
   do putStrLn "Calculo das raizes do polimomio a x^2 + b x + c"
      putStr "Indique o valor do ceoficiente a: "
      a <- getLine
      putStr "Indique o valor do ceoficiente b: "
      b <- getLine
      putStr "Indique o valor do ceoficiente c: "
      c <- getLine
      case (roots (read a, read b, read c)) of
         Nothing       -> putStrLn "Não há raizes reais."
         (Just (r1,r2)) -> putStrLn ("As raizes são "++(show r1)++" e "++(show r2))
```

`Float` é instância da classe `Read`

Input / Output

Uma maneira alternativa é usar a função **`readIO`** do Prelude

```
readIO :: Read a => String -> IO a
```

equivalente a `(return . read)`

```
calcROOTS :: IO ()
calcROOTS =
   do putStrLn "Calculo das raizes do polimomio a x^2 + b x + c"
      putStr "Indique o valor do ceoficiente a: "
      a <- getLine
      a1 <- readIO a
      putStr "Indique o valor do ceoficiente b: "
      b <- getLine
      b1 <- readIO b
      putStr "Indique o valor do ceoficiente c: "
      c <- getLine
      c1 <- readIO c
      case (roots  (a1,b1,c1)) of
         Nothing       -> putStrLn "Nao ha' raizes reais"
         (Just (r1,r2)) -> putStrLn ("As raizes sao "++(show r1)
                                     ++" e "++(show r2))
```


Input / Output

Exemplo: Carregar e descarregar uma base de dados de notas em ficheiro.

```
type Notas = [(Integer,String,Int)]
```

Ficheiro de texto

```
12345  Ana  16
33333  Nuno 12
11111  Rui  18
22222  Ines 15
```

```
leFich :: IO ()
leFich = do file <- dialogo "Qual o nome do ficheiro ? "
            s <- readFile file
            let l = map words (lines s)
            print (geraNotas l)

geraNotas :: [[String]] -> Notas
geraNotas ([x,y,z]:t) = (read x, y, read z):(geraNotas t)
geraNotas _ = []
```

```
escFich :: Notas -> IO ()
escFich notas = do file <- dialogo "Qual o nome do ficheiro ? "
                   writeFile file (geraStr notas)

geraStr :: Notas -> String
geraStr [] = ""
geraStr ((x,y,z):t) = (show x) ++ ('\t':y) ++ ('\t':(show z)) ++ "\n" ++ (geraStr t)
```


Exercício

Implementar um jogo de adivinha com as seguintes regras:

- É gerado um número inteiro aleatório entre 1 e n.
- O jogador tenta adivinhar o número e o computador responde se o número é baixo, se o número é alto, ou se acertou, contabilizando o número de tentativas feitas pelo jogador até acertar.

Para gerar o número aleatório vai ser preciso importar a biblioteca **`System.Random`**, onde está a classe **`Random`** (dos tipos para os quais é possível gerar valores aleatórios), da qual `Int` é uma instância.

A função da classe que nos interessa neste caso é

```
randomRIO :: Random a => (a,a) -> IO a
```

que gera um valor aleatório do tipo `a`, dentro de um intervalo.

Programas executáveis

- Para criar programas **executáveis** o compilador Haskell precisa de ter definido um módulo **`Main`** com uma função **`main`** que tem que ser de tipo **`IO a`** .
- A função main é o ponto de entrada no programa, pois é ela que é invocada quando o programa compilado é executado.
- A compilação de um programa Haskell, usando o GHC, pode ser feita executando no terminal do sistema operativo o seguinte comando:

`ghc -o` *nome_do_executável* **`--make`** *nome_do_ficheiro_do_módulo_principal*

Exercício: Crie um programa executável do jogo de adivinha que implementou.