

Programação Funcional CC

Lic. Ciências da Computação - 1º Ano

2007 / 2008

Maria João Frade (mjf@di.uminho.pt)

*Departamento de Informática
Universidade do Minho*

1

Um **programa** pode ser visto como algo que transforma informação



Existem 2 grandes classes de linguagens de programação:

Imperativas - um programa é uma seqüências de instruções (ou seja de “ordens”).
(ex: Pascal, C, Java, ...)

- difícil estabelecer uma relação precisa entre o input e o output e de raciocinar sobre os programas; ...
- + normalmente mais eficientes; ...

Declarativas - um programa é um conjunto de declarações que descrevem a relação entre o input e o output. (ex: Prolog, ML, Haskell, ...)

- + fácil de estabelecer uma relação precisa entre o input e o output e de raciocinar sobre os programas; ...
- normalmente menos eficientes (mas cada vez mais); ...

2

Exemplo: A função factorial é descrita matematicamente por

$$0! = 1$$
$$n! = n * (n-1)! , \text{ se } n > 0$$

Dois programas que fazem o cálculo do factorial de um número, implementados em:

C

```
int factorial(int n)
{ int i, r;

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

Haskell

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

Qual é mais facil de entender ?

3



Na programação (funcional) faremos uma distinção clara entre três grandes grupos de conceitos:

Dados – Que tipo de informação é recebida e como ela se pode organizar por forma a ser processada de forma eficiente.

Operações – Os mecanismos para manipular os dados. As operações básicas e como contruir novas operações a partir de outras já existentes.

Cálculo – A forma como o processo de cálculo decorre.

A linguagem **Haskell** fornece uma forma rigorosa e precisa de descrever tudo isto.

4

Programa Resumido

Nesta disciplina estuda-se o paradigma funcional de programação, tendo por base a linguagem de programação **Haskell**.

- Programação funcional em Haskell.
 - **Conceitos fundamentais**: expressões, tipos, redução, funções e recursividade.
 - **Conceitos avançados**: funções de ordem superior, polimorfismo, tipos indutivos, classes, modularidade e monades.
- Estruturas de dados e algoritmos.
- Tipos abstractos de dados.

5

Bibliografia

- **Fundamentos da Computação, Livro II: Programação Funcional**.
José Manuel Valença e José Bernardo Barros. Universidade Aberta, 1999.
- **Introduction to Functional Programming using Haskell**.
Richard Bird. Prentice-Hall, 1998.
- **Haskell: the craft of functional programming**.
Simon Thompson. Addison-Wesley.
- **A Gentle Introduction to Haskell**.
Paul Hudak, John Peterson and Joseph Fasel.
- **Apontamentos da aulas teóricas e fichas práticas**.

Disponíveis em www.di.uminho.pt/~mjf/PFcc/2007-08

Acessíveis a partir de lcc.di.uminho.pt

6

Programação Funcional CC

8 ECTS = 224 horas de trabalho
(150 horas de trabalho independente)

2 Teóricas

1 Teórico-Prática

1 Prática Laboratorial

7

Critérios de Avaliação

A avaliação tem uma **componente teórica** e uma **componente prática**, ambas **obrigatórias**.

$$\text{Nota Final} = 0.4 \text{ NP} + 0.6 \text{ NT}$$

sendo:

NT a nota teórica ($\text{NT} \geq 9.5$, obrigatoriamente), obtida através da realização de uma prova individual escrita;

NP a nota prática ($\text{NP} \geq 9.5$, obrigatoriamente), resultante da realização de um teste laboratorial e um trabalho prático.

O trabalho prático são realizados em grupo (de 3 alunos), mas a nota prática é individual.

8

Datas previstas para as avaliações

Teste laboratorial = semana de **26 de Novembro**

Trabalho prático = **entrega** na semana **14 de Janeiro**
apresentação na semana **21 de Janeiro**

Teste final = **29 de Janeiro** (Manhã)

9

O Paradigma Funcional de Programação

Haskell

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

As equações que são usadas na definição da função `fact` são **equações matemáticas**. Elas indicam que o lado esquerdo e direito têm o mesmo valor.

C

```
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.

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 valor anterior mais 1). Portanto `i` é redefinido.

Porque `=` em Haskell significa **“é, por definição, igual a”**, e não é possível redefinir, o que fazemos é raciocinar sobre equações matemáticas.

É, portanto, muito mais fácil do que raciocinar sobre programas funcionais do que sobre programas imperativos.

10

O Paradigma Funcional de Programação

- Um **programa** é um conjunto de definições.
- Uma **definição** associa um *nome* a um *valor*.
- **Programar** é definir estruturas de dados e funções para resolver um dado problema.
- O **interpretador** (da linguagem funcional) actua como uma máquina de calcular:

lê uma expressão, calcula o seu valor e mostra o resultado

Exemplo: Um programa para converter valores de temperaturas em graus *Celcius* para graus *Fahrenheit*, e de graus *Kelvin* para graus *Celcius*.

```
celFar c = c * 1.8 + 32
kelCel k = k - 273
```

Depois de carregar este programa no interpretador Haskell, podemos fazer os seguintes testes:

```
> celFar 25
77.0
> kelCel 0
-273
>
```

11

- A um conjunto de associações *nome-valor* dá-se o nome de **ambiente** ou **contexto** (ou *programa*).
- As expressões são avaliadas no âmbito de um contexto e podem conter ocorrências dos nomes definidos nesse contexto.
- O interpretador usa as **definições** que tem no contexto (programa) **como regras de cálculo**, para simplificar (calcular) o valor de uma expressão.

Exemplo: Este programa define três funções de conversão de temperaturas.

```
celFar c = c * 1.8 + 32
kelCel k = k - 273
kelFar k = celFar (kelCel k)
```

No interpretador ...

```
> kelFar 300
80.6
```

É calculado pelas regras estabelecidas pelas definições fornecidas pelo programa.

```
kelFar 300 ⇒ celFar (kelCel 300)
            ⇒ (kelCel 300) * 1.8 + 32
            ⇒ (300 - 273) * 1.8 + 32
            ⇒ 27 * 1.8 + 32
            ⇒ 80.6
```

12

Transparência Referencial

- No paradigma funcional, as expressões:
 - são a representação concreta da informação;
 - podem ser associadas a nomes (definições);
 - denotam valores que são determinados pelo interpretador da linguagem.
- No âmbito de um dado contexto, todos os nomes que ocorrem numa expressão têm um **valor único** e **imutável**.
- O valor de uma expressão depende *unicamente* dos valores das sub-expressões que a constituem, e essas podem ser substituídas por outras que possuam o mesmo valor.

A esta característica dá-se o nome de **transparência referencial**.

13

Linguagens Funcionais

- O nome de *linguagens funcionais* advém do facto de estas terem como operações básicas a **definição de funções** e a **aplicação de funções**.
 - Nas linguagens funcionais as 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.
- Isto dá às linguagens funcionais uma **grande flexibilidade, capacidade de abstracção e modularização do processamento de dados**.
- As linguagens funcionais fornecem um alto nível de abstracção, o que faz com que os programas funcionais sejam mais **concisos**, mais **fáceis de entender / manter** e mais **rápidos de desenvolver** do que programas imperativos.
 - No entanto, em certas situações, os programas funcionais podem ser mais penalizadores em termos de eficiência.

14

Um pouco de história ...

1960s **Lisp** (*untyped, not pure*)

1970s **ML** (*strongly typed, type inference, polymorphism*)

1980s **Miranda** (*strongly typed, type inference, polymorphism, lazy evaluation*)

1990s **Haskell** (*strongly typed, type inference, polymorphism, lazy evaluation, ad-hoc polymorphism, monadic IO*)

15

Haskell

- O Haskell é uma linguagem puramente funcional, fortemente tipada, e com um sistema de tipos extremamente evoluído.
- A linguagem usada neste curso é o **Haskell 98**.
- Exemplos de interpretadores e um compilador para a linguagem Haskell 98:
 - **Hugs** *Haskell User's Gofer System*
 - **GHC** *Glasgow Haskell Compiler* (é o que vamos usar ...)

www.haskell.org

16

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 I/O 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 lazy functional languages.

(The Haskell 98 Report)

17

Valores & Expressões

Os **valores** são as entidades básicas da linguagem Haskell. São os elementos atômicos.

As **expressões** são obtidas aplicando funções a valores ou a outras expressões.

O interpretador Haskell actua como uma *calculadora* (“read - evaluate - print loop”):

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

Exemplos:

```
> 5
5
> 3.5 + 6.7
10.2
> 2 < 35
True
> not True
False
> not ((3.5+6.7) > 23)
True
```

18

Tipos

Os tipos servem para **classificar** entidades (de acordo com as suas características).

Em Haskell *toda a expressão tem um tipo associado*.

e :: T

significa que a expressão **e** *tem* tipo **T**
é do tipo

Informalmente, podemos associar a noção de “*tipo*” à noção de “*conjunto*”, e a noção de “*ter tipo*” à noção de “*pertença*”.

Exemplos:

58	::	Int	Inteiro
'a'	::	Char	Caracter
[3,5,7]	::	[Int]	Lista de inteiros
(8, 'b')	::	(Int, Char)	Par com um inteiro e um caracter

O Haskell é uma linguagem **fortemente tipada**, com um sistema de tipos muito evoluído (como veremos). Em Haskell, a verificação de tipos é feita durante a compilação.

19

Tipos Básicos

Bool	Booleanos:	True, False
Char	Caracteres:	'a', 'b', 'A', '1', '\n', '2', ...
Int	Inteiros de tamanho limitado:	1, -3, 234345, ...
Integer	Inteiros de tamanho ilimitado:	2, -7, 75756850013434682, ...
Float	Números de vírgula flutuante:	3.5, -6.53422, 51.2E7, 3e4, ...
Double	Núm. vírg. flut. de dupla precisão:	3.5, -6.5342, 51.2E7, ...
()	<i>Unit</i>	() é o seu único elemento do tipo <i>Unit</i> .

20

Tipos Compostos

Produtos Cartesianos (T_1, T_2, \dots, T_n)

(T_1, T_2, \dots, T_n) é o tipo dos tuplos com o 1º elemento do tipo T_1 , 2º elemento do tipo T_2 , etc.

Exemplos:

```
(1, 5) :: (Int, Int)
('a', 6, True) :: (Char, Int, Bool)
```

Listas $[T]$

$[T]$ é o tipo da listas cujos elementos *são todos* do tipo T .

Exemplos:

```
[2, 5, 6, 8] :: [Integer]
['h', 'a', 's'] :: [Char]
[3.5, 86.343, 1.2] :: [Float]
```

Funções $T_1 \rightarrow T_2$

$T_1 \rightarrow T_2$ é o tipo das funções que *recebem* valores do tipo T_1 e *devolvem* valores do tipo T_2 .

Exemplos:

```
not :: Bool -> Bool
ord :: Char -> Int
```

21

Funções

A operação mais importante das funções é a sua **aplicação**.

Se $f :: T_1 \rightarrow T_2$ e $a :: T_1$ então $f a :: T_2$

Exemplos:

```
> not True
False :: Bool

> ord 'a'
97 :: Int

> ord 'A'
65 :: Int

> chr 97
'a' :: Char
```

Preservação de Tipos

O tipo de cada expressão é preservado ao longo do processo de cálculo.

Qual será o tipo de `chr` ?

Novas definições de funções deverão que ser escritas num ficheiro, que depois será carregado no interpretador.

22

Definições

Uma definição **associa** um nome a uma expressão.

`nome = expressão`

nome tem que ser uma palavra começada por letra minúscula.

A definição de funções pode ainda ser feita por um conjunto de **equações** da forma:

`nome arg1 arg2 ... argn = expressão`

Quando se define uma função podemos incluir *informação sobre o seu tipo*. No entanto, essa informação **não é obrigatória**.

Exemplos:

```
pi = 3.1415
areaCirc x = pi * x * x
areaQuad = \x -> x*x
areaTri b a = (b*a)/2
volCubo :: Float -> Float
volCubo y = y * y * y
```

23

Pólimorfismo

O tipo de cada função é **inferido automaticamente** pelo interpretador.

Exemplo:

Para a função *g* definida por: `g x = not (65 > ord x)`

O tipo inferido é `g :: Char -> Bool`

Porquê ?

Mas, há funções às quais é possível associar *mais do que um* tipo concreto.

Exemplos:

```
id x = x
```

```
nl y = '\n'
```

O que fazem estas funções ?

Qual será o seu tipo ?

24

O problema é resolvido recorrendo a **variáveis de tipo**.

Uma variável de tipo representa um tipo qualquer.

```
id :: a -> a
nl :: a -> Char
```

Em Haskell:

- As variáveis de tipo representam-se por nomes começados por letras minúsculas (normalmente a, b, c, ...).
- Os tipos concretos usam nomes começados por letras maiúsculas (ex: Bool, Int, ...).

Quando as funções são usadas, as variáveis de tipos são substituídas pelos tipos concretos adequados.

Exemplos:

```
id True
id 'a'
nl False
nl (volCubo 3.2)
```

```
id :: Bool -> Bool
id :: Char -> Char
nl :: Bool -> Char
nl :: Float -> Char
```

25

Funções cujos tipos têm variáveis de tipo são chamadas **funções polimórficas**.

Um tipo pode conter diferentes variáveis de tipo.

Exemplo:

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

Inferência de tipos

O tipo de cada função é inferido automaticamente pelo compilador de Haskell.

O compilador infere sempre o *tipo mais preciso* de qualquer expressão (o seu tipo principal).

É possível associar a uma função um tipo *mais específico* do que o tipo inferido automaticamente.

Exemplo:

```
seg :: (Bool,Int) -> Int
seg (x,y) = y
```

26

O utilizador pode fazer dois tipos de pedidos ao interpretador **ghci**:

- [Calcular o valor de uma expressão.](#)

```
Prelude> 3+5
8
Prelude> (5>=7) || (3^2 == 9)
True
Prelude> fst (40/2, 'A')
20.0
Prelude> pi
3.141592653589793
Prelude> aaa

<interactive>:1: Variable not in scope: `aaa'
Prelude>
```

- [Executar um comando.](#)

- Os comandos do **ghci** começam sempre por dois pontos (:).
- O comando **:?** lista todos os comandos existentes

```
Prelude> :?
Commands available from the prompt:
```

...

29

Alguns comandos úteis:

:quit ou **:q** termina a execução do **ghci**.

:type ou **:t** indica o tipo de uma expressão.

```
Prelude> :type (2>5)
(2>5) :: Bool
Prelude> :t not
not :: Bool -> Bool
Prelude> :q
Leaving GHCi.
```

:load ou **:l** carrega o programa (o módulo) que está num dado ficheiro.

Exemplo: Considere o seguinte programa guardado no ficheiro `Temp.hs`

```
module Temp where

celFar c = c * 1.8 + 32

kelCel k = k - 273

kelFar k = celFar (kelCel k)
```

Os programas em Haskell têm normalmente extensão **.hs** (de *haskell script*)

30

Depois de carregar um módulo, os nomes definidos nesse módulo passam a estar disponíveis no ambiente de interpretação

```
Prelude> kelCel 300
<interactive>:1: Variable not in scope: `kelCel'
Prelude> :load Temp
Compiling Temp          ( Temp.hs, interpreted )
Ok, modules loaded: Temp.
*Temp> kelCel 300
27
*Temp>
```

Inicialmente, apenas as declarações do módulo Prelude estão no ambiente de interpretação. Após o carregamento do ficheiro Temp.hs, ficam no ambiente todas as definições feitas no módulo Temp e as definições do Prelude.

31

Um **módulo** constitui um *componente de software* e dá a possibilidade de gerar bibliotecas de funções que podem ser **reutilizadas** em diversos programas Haskell.

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

Para se utilizarem declarações feitas noutros módulos, que não o Prelude, é necessário primeiro fazer a sua importação através da instrução:

```
import Nome_do_módulo
```

Exemplo.hs

```
module Exemplo where

import Char

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

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

32

Comentários

É possível colocar **comentários** num programa Haskell de duas formas:

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

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

```
module Temp where

-- de Celcius para Farenheit
celFar c = c * 1.8 + 32

-- de Kelvin para Celcius
kelCel k = k - 273

-- de Kelvin para Farenheit
kelFar k = celFar (kelCel k)

{- dado valor da temperatura em Kelvin, retorna o triplo com
o valor da temperatura em Kelvin, Celcius e Farenheit -}

kelCelFar k = (k, kelCel k, kelFar k)
```

33

As funções `test` e `test'` são muito parecidas mas há uma diferença essencial:

```
test (x,y) = [ (not x), (y || x), (x && y) ]
test' x y = [ (not x), (y || x), (x && y) ]
```

Têm tipos diferentes !

A função `test` recebe **um único argumento** (que é um par de booleanos) e devolve uma lista de booleanos.

```
test :: (Bool,Bool) -> [Bool]
```

```
> test (True,False)
```

A função `test'` recebe **dois argumentos**, cada um do tipo `Bool`, e devolve uma lista de booleanos.

```
test' :: Bool -> Bool -> [Bool]
```

```
> test' True False
```

A função `test'` recebe um valor de cada vez. Realmente, o seu tipo é:

```
test' :: Bool -> (Bool -> [Bool])
```

```
> (test' True) False
```

Mas os parentesis podem ser dispensados !

34

- **O tipo função associa à direita.**

Isto é, `f :: T1 -> T2 -> ... -> Tn -> T`

é uma forma abreviada de escrever

```
f :: T1 -> (T2 -> (... -> (Tn -> T)...))
```

- **A aplicação de funções é associativa à esquerda.**

Isto é, `f x1 x2 ... xn`

é uma forma abreviada de escrever

```
(...((f x1) x2) ...) xn
```

35

Exercício:

Considere a seguinte declaração das funções `fun1`, `fun2` e `fun3`.

```
fun1 (x,y) = (not x) || y
fun2 a b = (a||b, a&&b)
fun3 x y z = x && y && z
```

Qual será o tipo de cada uma destas funções ?

Dê exemplos da sua invocação.

36

Lista e String

[a] é o tipo das listas cujos elementos *são todos* do tipo **a**.

```
Exemplos: [2,5,6,8] :: [Integer]
           [(1+3,'c'),(8,'A'),(4,'d')] :: [(Int,Char)]
           [3.5, 86.343, 1.2*5] :: [Float]
           ['0','1','a'] :: [Char]
```

Atenção! `['A', 4, 3, 'C']` **Não são listas bem formadas**, porque os seus elementos não têm todos o mesmo tipo!
`[(1,5), 9, (6,7)]`

String O Haskell tem pré-definido o tipo **String** como sendo **[Char]**.

Os valores do tipo String também se escrevem de forma abreviada entre “ ”.

Exemplo: `“haskell”` é equivalente a `['h','a','s','k','e','l','l']`

```
> “Ola” == ['O','l','a']
True
```

37

Algumas funções sobre listas definidas no Prelude.

head :: [a] -> a dá o primeiro elemento da lista (a cabeça da lista).

tail :: [a] -> [a] dá a lista sem o primeiro elemento (a cauda da lista).

take :: Int -> [a] -> [a] dá um segmento inicial de uma lista.

drop :: Int -> [a] -> [a] dá um segmento final de uma lista.

reverse :: [a] -> [a] calcula a lista invertida.

last :: [a] -> a dá o último elemento da lista.

Exemplos:

```
Prelude> head [3,4,5,6,7,8,9]
3
Prelude> tail ['a','b','c','d']
['b','c','d']
Prelude> take 3 [3,4,5,6,7,8,9]
[3,4,5]
```

```
Prelude> drop 3 [3,4,5,6,7,8,9]
[6,7,8,9]
Prelude> reverse [3,4,5,6,7,8,9]
[9,8,7,6,5,4,3]
Prelude> last ['a','b','c','d']
'd'
```

38

Funções sobre String definidas no Prelude.

words :: String -> [String] dá a lista de palavras de um texto.

unwords :: [String] -> String constrói um texto a partir de uma lista de palavras.

lines :: String -> [String] dá a lista de linhas de um texto (i.e. parte pelo '\n').

Exemplos:

```
Prelude> words "aaaa bbbb cccc\tdddd eeee\nffff gggg hhhh"  
["aaaa","bbbb","cccc","dddd","eeee","ffff","gggg","hhhh"]
```

```
Prelude> unwords ["aaaa","bbbb","cccc","dddd","eeee","ffff","gggg","hhhh"]  
"aaaa bbbb cccc dddd eeee ffff gggg hhhh"
```

```
Prelude> lines "aaaa bbbb cccc\tdddd eeee\nffff gggg hhhh"  
["aaaa bbbb cccc\tdddd eeee","ffff gggg hhhh"]
```

```
Prelude> reverse "programacao funcional"  
"lanoicnuf oacamargorp"
```

39

Listas por Compreensão

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

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

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

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

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

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

$\{(x,y) \mid x \in \{3,4,5\} \wedge y \in \{9,10\}\}$

`[(x,y) | x <- [3,4,5], y <- [9,10]]`
= [(3,9),(3,10),(4,9),(4,10),(5,9),(5,10)]

40

Listas infinitas

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

$\{x^3 \mid x \in \mathbb{N} \wedge \text{par}(x)\}$

`[x^3 | x <- [0..], even x]` = [0, 8, 46, 216, ...]

Mais exemplos:

```
Prelude> ['A'..'Z']
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
Prelude> ['A', 'C'..'X']
"ACEGIKMOQSUW"
Prelude> [50, 45..(-20)]
[50, 45, 40, 35, 30, 25, 20, 15, 10, 5, 0, -5, -10, -15, -20]
Prelude> drop 20 ['a'..'z']
"vwxyz"
Prelude> take 10 [3, 3..]
[3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
```

41

Equações e Funções

Uma função pode ser definida por equações que relacionam os seus argumentos com o resultado pretendido.

Exemplos:

```
triplo x = 3 * x
dobro y = y + y
perimCirc r = 2*pi*r
perimTri x y z = x+y+z
minimo x y = if x>y then y else x
```

As equações definem regras de cálculo para as funções que estão a ser definidas.

`nome arg1 arg2 ... argn = expressão`

Nome da função
(iniciada por letra minúscula).

Argumentos da função.
Cada argumento é um **padrão**.
(cada variável não pode ocorrer mais do que uma vez)

O tipo da função é *inferido* tendo por base que ambos os lados da equação têm que ter o mesmo tipo.

42

Padrões (patterns)

Um **padrão** é uma **variável**, uma **constante**, ou um “**esquema**” de um valor atômico (isto é, o resultado de aplicar construtores básicos dos valores a outros padrões).

No Haskell, um padrão **não** pode ter variáveis repetidas (*padrões lineares*).

Exemplos:

Padrões	Tipos	Não padrões
x True 4 (x, y, (True, b)) ('A', False, x) [x, 'a', y]	a Bool Int (a, b, (Bool, c)) (Char, Bool, a) [Char]	[x, 'a', 1] (4*6, y) Porquê ?

Quando não nos interessa dar nome a uma variável, podemos usar `_` que representa uma *variável anônima nova*.

Exemplos:

```
snd (_, x) = x  
segundo (_, y, _) = y
```

43

Exemplos:

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

outro modo seria

```
soma w z = ((fst w)+z, (snd w)+z)
```

Qual é mais legível ?

```
exemplo :: (Bool, Float) -> ((Float, Int), Float) -> Float  
exemplo (True, y) ((x, _), w) = y*x + w  
exemplo (False, y) _ = y
```

em alternativa, poderíamos ter

```
exemplo a b = if (fst a) then (snd a)*(fst (fst b)) + (snd b)  
                else (snd a)
```

44

Redução

O cálculo do valor de uma expressão é feito usando as equações que definem as funções como regras de cálculo.

Uma **redução** é um passo do processo de cálculo (é usual usar o símbolo \Rightarrow denotar esse passo)

Cada redução resulta de substituir a *instância* do lado esquerdo da equação (o **redex**) pelo respectivo lado direito (o **contractum**).

Exemplos: Relembre as seguintes funções

```
triplo x = 3 * x
dobro y = y + y
snd (_,x) = x
nl x = '\n'
```

Exemplos: `triplo 7` \Rightarrow `3*7` \Rightarrow `21`

A instância de `(triplo x)` resulta da *substituição* `[7/x]`.

`snd (9,8)` \Rightarrow `8`

A instância de `snd (_,x)` resulta da *substituição* `[9/_,8/x]`.

45

A expressão `dobro (triplo (snd (9,8)))` pode reduzir de três formas distintas:

`dobro (triplo (snd (9,8)))` \Rightarrow `dobro (triplo 8)`

`dobro (triplo (snd (9,8)))` \Rightarrow `dobro (3*(snd (9,8)))`

`dobro (triplo (snd (9,8)))` \Rightarrow `(triplo (snd (9,8)))+(triplo (snd (9,8)))`

A estratégia de redução usada para o cálculo das expressões é uma característica essencial de uma linguagem funcional.

O **Haskell** usa a estratégia **lazy evaluation** (*call-by-name*), que se caracteriza por escolher para reduzir sempre o redex mais externo. Se houver vários redexes ao mesmo nível escolhe o redex mais à esquerda (*outermost; leftmost*).

Uma outra estratégia de redução conhecida é a **eager evaluation** (*call-by-value*), que se caracteriza por escolher para reduzir sempre o redex mais interno. Se houver vários redexes ao mesmo nível escolhe o redex mais à esquerda (*innermost; leftmost*).

46

Lazy Evaluation (*call-by-name*)

```
dobro (triplo (snd (9,8))) ⇒ (triplo (snd (9,8)))+(triplo (snd (9,8)))
                          ⇒ (3*(snd (9,8))) + (triplo (snd (9,8)))
                          ⇒ (3*(snd (9,8))) + (3*(snd (9,8)))
                          ⇒ (3*8) + (3*(snd (9,8)))
                          ⇒ 24 + (3*(snd (9,8)))
                          ⇒ 24 + (3*8)
                          ⇒ 24 + 24
                          ⇒ 48
```

Com a estratégia *lazy* os parametros das funções só são calculados se o seu valor for mesmo necessário.

```
n1 (triplo (dobro (7*45))) ⇒ '\n'
```

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.

```
n1 (3/0) ⇒ '\n'
```

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

47

Podemos definir uma função recorrendo a várias equações.

Exemplo:

```
h :: (Char, Int) -> Int
h ('a', x) = 3*x
h ('b', x) = x+x
h (_, x) = x
```

Todas as equações têm que ser bem tipadas e de tipos coincidentes.

Cada equação é usada como regra de redução. Quando uma função é aplicada a um argumento, a equação que é selecionada como regra de redução é a **1ª equação** (a contar de cima) cujo **padrão** que tem como argumento **concorda** com o argumento actual (*pattern matching*).

Exemplos:

```
h ('a', 5) ⇒ 3*5 ⇒ 15
h ('b', 4) ⇒ 4+4 ⇒ 8
h ('B', 9) ⇒ 9
```

Note: Podem existir *várias* equações com padrões que concordam com o argumento actual. Por isso, a ordem das equações é importante, pois define uma prioridade na escolha da regra de redução.

O que acontece se alterar a ordem das equações que definem h ?

48

Funções Totais & Funções Parciais

Uma função diz-se **total** se está definida para todo o valor do seu domínio.

Uma função diz-se **parcial** se há valores do seu domínio para os quais ela não está definida (isto é, não é capaz de produzir um resultado no conjunto de chegada).

Exemplos:

```
conjuga :: (Bool,Bool) -> Bool
conjuga (True,True) = True
conjuga (x,y) = False
```

Função total

```
parc :: (Bool,Bool) -> Bool
parc (True,False) = False
parc (True,x) = True
```

Função parcial

Porquê ?

49

Tipos Sinónimos

O Haskell pode renomear tipos através de declarações da forma:

```
type Nome p1 ... pn = tipo
```

parâmetros (*variáveis de tipo*)

Exemplos:

```
type Ponto = (Float,Float)
type ListaAssoc a b = [(a,b)]
```

Note que não estamos a criar tipos novos, mas apenas nomes novos para tipos já existentes. Esses nomes devem contribuir para a compreensão do programa.

Exemplo:

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

O tipo **String** é outro exemplo de um tipo sinónimo, definido no Prelude.

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

50

Definições Locais

Uma definição associa um nome a uma expressão.

Todas as definições feitas até aqui podem ser vistas como **globais**, uma vez que elas são visíveis no *módulo* do programa aonde estão. Mas, muitas vezes é útil reduzir o âmbito de uma declaração.

Em Haskell há duas formas de fazer definições **locais**: utilizando expressões **let ... in** ou através de cláusulas **where** junto da definição equacional de funções.

Exemplos:

```
let c = 10
    (a,b) = (3*c, f 2)
    f x = x + 7*c
in f a + f b
```

⇒ 242

Porquê ?

```
testa y = 3 + f y + f a + f b
  where c = 10
        (a,b) = (3*c, f 2)
        f x = x + 7*c
```

```
> testa 5
320
> c
Variable not in scope: `c`
> f a
Variable not in scope: `f`
Variable not in scope: `a`
```

As declarações locais podem ser de funções e de identificadores (fazendo uso de padrões).

51

Layout

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

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

Regras fundamentais:

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

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

Exemplo:

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

52

Operadores

Operadores infixos como o $+$, $*$, $\&\&$, \dots , não são mais do que funções.

Um operador infixos pode ser usado como uma função vulgar (i.e., usando notação prefixa) se estiver entre parentesis.

Exemplo: $(+)$ $2\ 3$ é equivalente a $2+3$

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

Podem-se definir novos operadores infixos.

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

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

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

3 ``mod`` 2 é equivalente a `mod 3 2`

53

Cada operador tem uma **prioridade** e uma **associatividade** estipulada.

Isto faz com que seja possível evitar alguns parentesis.

Exemplo: $x + y + z$ é equivalente a $(x + y) + z$
 $x + 3 * y$ é equivalente a $x + (3 * y)$

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

Exemplo: $f\ x * y$ é equivalente a $(f\ x) * y$

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

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

54

Funções com Guardas

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

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

Exemplos:

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

é equivalente a

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

ou a

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

otherwise é equivalente a **True**.

55

Exemplo: Raizes reais do polinómio $ax^2 + bx + c$

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

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

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

```
> raizes (2,10,3)
(-0.320550528229663,-4.6794494717703365)
> raizes (2,3,4)
*** Exception: raizes imaginarias
```

56

Listas

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

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

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

- a lista vazia []
- o construtor (:), que é um operador infixado que dado um elemento x de tipo a e uma lista l de tipo [a], constrói uma nova lista com x na 1ª posição seguida de l.

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

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

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

57

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

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

Qual o tipo destas funções ?

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

As funções são totais ou parciais?

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

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

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

Em soma3 a ordem das equações é importante ? Porquê ?

58

Recorrência

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

Temos dois casos:

- Se a lista for vazia o seu comprimento é zero.
- Se a lista não for vazia o seu comprimento é um mais o comprimento da cauda da lista.

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

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

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

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

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

59

Mais alguns exemplos de funções já definidas no módulo Prelude:

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

Qual o tipo destas funções ?

São totais ou parciais ?

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

Podemos trocar a ordem das equações ?

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

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

60

Considere a função `zip` já definida no Perlude:

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

Qual o seu tipo ? É total ou parcial ?
Podemos trocar a ordem das equações ?
Podemos dispensar alguma equação ?
Será que podemos definir `zip` com menos equações ?

Exercícios:

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

`zip [1,2] "LCC"`

- Defina a função que faz o "zip" de 3 listas.
- Defina a função `unzip :: [(a,b)] -> ([a],[b])`

61

Padrões sobre números naturais.

O Haskell aceita como um padrão sobre números naturais, expressões da forma:

(*variável* + *número_natural*)

Exemplos:

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

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

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

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

Atenção: expressões como

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

não são padrões !

62

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

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

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

O que fazem estas funções ?

Qual o seu tipo ?

Estas funções serão totais ?

Trocando a ordem das equações, será que obtemos a mesma função ?

63

As funções **take** e **drop** estão pré-definidas no **Prelude** da seguinte forma:

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

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

Estas funções serão totais ?

Trocando a ordem das equações, será que obtemos a mesma função ?

Defina funções equivalentes utilizando padrões de números naturais.

64

nome@padrão

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

Exemplos:

A função `fun :: (Int,String) -> (Char,(Int,String))`

pode ser definida, equivalentemente, por:

```
fun (n,(x:xs)) = (x,(n,(x:xs)))
```

ou `fun par@(n,(x:xs)) = (x,par)`

ou `fun (n,(x:xs)) = let par = (n,(x:xs))
in (x,par)`

```
{- Esta função vai retirando os elementos de uma lista até  
encontrar um elemento não positivo -}
```

```
dropWhilePos [] = []  
dropWhilePos lis@(x:xs) | x > 0 = dropWhilePos xs  
| otherwise = lis
```

65

Algoritmos de Ordenação

A ordenação de um conjunto de valores é um problema muito frequente, e muito útil na organização de informação.

Para o problema de ordenação de uma lista de valores, existem diversos algoritmos:

- *Insertion Sort*
- *Quick Sort*
- *Merge Sort*
- ...

Vamos apresentar estes algoritmos, para *ordenar uma lista de valores por ordem crescente*, de acordo com os operadores relacionais `<`, `<=`, `>`, e `>=` (que implicitamente assumimos estarem definidos para os tipos desses valores).

66

Insertion Sort

Algoritmo:

1. Seleciona-se a cabeça da lista.
2. Ordena-se a cauda da lista.
3. Insere-se a cabeça da lista na cauda ordenada, de forma a que a lista resultante continue ordenada.

```
isort [] = []  
isort (x:xs) = insert x (isort xs)
```

```
insert x [] = [x]  
insert x (y:ys) = if x < y then x:y:ys  
                  else y:(insert x ys)
```

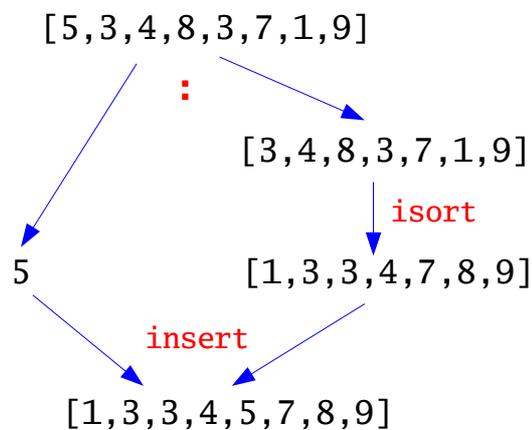
A função `insert` (que faz a inserção ordenada) é o núcleo deste algoritmo.

```
isort [3,5,6,2,7,5,8] ⇒ insert 3 (isort [5,6,2,7,5,8])  
                    ⇒ ... ⇒ insert 3 [2,5,5,6,7,8]  
                    ⇒ ... ⇒ [2,3,5,5,6,7,8]
```

67

Insertion Sort

Exemplo: Esquema do cálculo de `(isort [5,3,4,8,3,1,9])`



68

Quick Sort

Algoritmo:

1. Seleciona-se a cabeça da lista (como *pivot*) e parte-se o resto da lista em duas sublistas: uma com os elementos inferiores ao pivot, e outra com os elementos não inferiores.
2. Estas sublistas são ordenadas.
3. Concatena-se as sublistas ordenadas, de forma adequada, conjuntamente com o pivot.

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

Esta versão do qsort é pouco eficiente ...

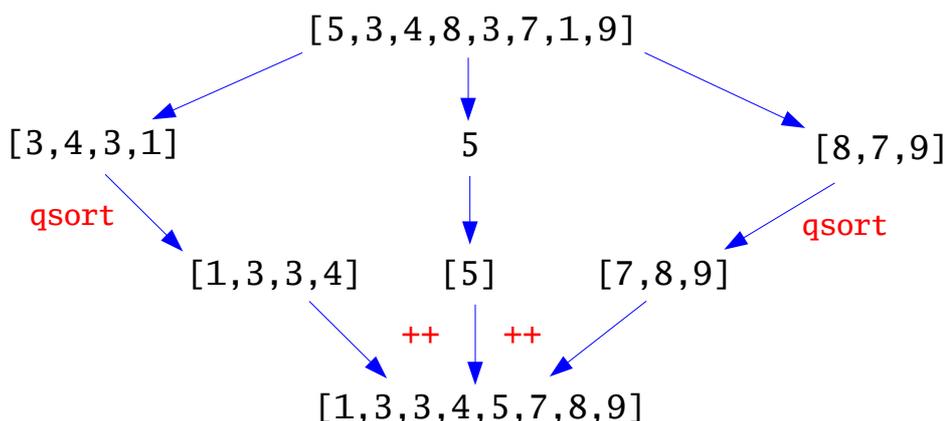
Quantas travessias da lista se estão a fazer para partir a lista ?

```
qsort [5,3,4,8,3,7,1,9] =>  
    ... => (qsort [3,4,3,1])++[5]++(qsort [8,7,9])  
=> ... => [1,3,3,4] ++ [5] ++ [7,8,9]  
=> ... => [1,3,3,4,5,7,8,9]
```

69

Quick Sort

Exemplo: Esquema do cálculo de `(qsort [5,3,4,8,3,1,9])`



Uma *versão mais eficiente* (fazendo a partição da lista numa só passagem), pode ser:

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

```
quicksort [] = []  
quicksort (x:xs) = let (l1,l2) = parte x xs  
                    in (quicksort l1)++[x]++(quicksort l2)
```

70

Merge Sort

Algoritmo:

1. Parte-se a lista em duas sublistas de tamanho igual (ou quase).
2. Ordenam-se as duas sublistas.
3. Fundem-se as sublistas ordenadas, de forma a que a lista resultante fique ordenada.

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

```
msort [] = []
msort [x] = [x]
msort xs = merge (msort xs1) (msort xs2)
  where
    k    = (length xs) `div` 2
    xs1  = take k xs
    xs2  = drop k xs
```

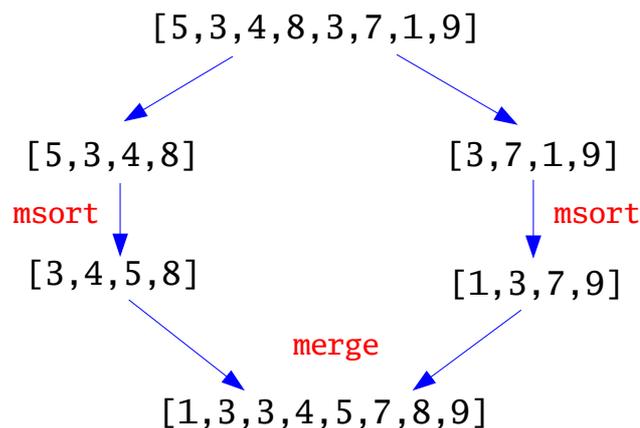
Esta versão do msort é muito pouco eficiente ...

Quantas travessias da lista se está a fazer para partir a lista em duas ?

71

Merge Sort

Exemplo: Esquema do cálculo de `(msort [5,3,4,8,3,1,9])`



Uma *versão mais eficiente* (fazendo a partição da lista numa só passagem), pode ser:

```
split [] = ([],[])
split (x:xs) = let (l,r) = split xs
                in (x:r,l)
```

```
mergesort [] = []
mergesort [x] = [x]
mergesort l = merge (mergesort l1) (mergesort l2)
  where (l1,l2) = split l
```

72

Acumuladores

Considere a definição da função `factorial`.

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

O cálculo da factorial de um número positivo n é feito multiplicando n pelo factorial de $(n-1)$. A multiplicação fica *em suspenso* até que o valor de `fact (n-1)` seja sintetizado.

```
fact 3 => 3*(fact 2) => 3*(2*(fact 1)) => 3*(2*(1*(fact 0)))
=> 3*(2*(1*1)) => 6
```

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

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

```
fact 3 => factAc 1 3 => factAc (1*3) 2 => factAc (1*3*2) 1
=> factAc (1*2*3*1) 0 => 1*2*3*1 => 6
```

73

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

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

Exemplo:

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

Qual lhe parece mais natural ?

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

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

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

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

74

Considere a função que inverte uma lista.

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

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

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

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

```
reverse [1,2,3] => revAc [] [1,2,3] => revAc [1] [2,3]  
=> revAc [2,1] [3] => revAc [3,2,1] [] => [3,2,1]
```

75

Funções de Ordem Superior

Em Haskell, as funções são entidades de primeira ordem, isto é, as **funções** podem **ser passadas como parametro** e/ou **devolvidas como resultado** de outras funções

Exemplo: A função **app** tem como argumento uma função **f** de tipo **a->b**.

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

```
app fact (5,4) => (120,24)
```

```
app chr (65,70) => ('A', 'F')
```

Exemplo:

A função **mult** pode ser entendida como tendo **dois argumentos** de tipo **Int** e devolvendo um valor do tipo **Int**. Mas, na realidade, **mult** é uma função que recebe **um argumento** do tipo **Int** e devolve uma função de tipo **Int->Int**.

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

Em Haskell, todas a funções são unárias !

```
mult 2 5 ≡ (mult 2) 5 :: Int  
          (mult 2) :: Int -> Int
```

76

map

Podemos definir uma função de ordem superior que aplica uma função ao longo de uma lista:

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

Note que `(map f lista)` é equivalente a `[f x | x <- lista]`

Podemos definir as funções do slide anterior à custa da função `map`, fazendo:

```
distancias lp = map distOrigem lp
```

```
minusculas s = map toLower s
```

```
triplica xs = map (3*) xs
```

```
factoriais ns = map fact ns
```

Ou então,

Porquê ?

```
distancias = map distOrigem
```

```
minusculas = map toLower
```

```
triplica = map (3*)
```

```
factoriais = map fact
```

79

filter

Considere as seguintes funções:

```
aprova :: [Int] -> [Int]
aprova [] = []
aprova (x:xs) = if (10<=x) then x:(aprova xs)
                else (aprova xs)
```

```
filtraDigitos :: String -> String
filtraDigitos [] = []
filtraDigitos (c:cs)
  | isDigit c = c:(filtraDigitos cs)
  | otherwise = filtraDigitos cs
```

```
primQuad :: [Ponto] -> [Ponto]
primQuad [] = []
primQuad ((x,y):ps) | x>0 && y>0 = (x,y):(primQuad ps)
                    | otherwise = primQuad ps
```

Todas estas funções têm um *padrão de computação* comum:

dada uma lista, geram uma nova lista com os elementos da lista que satisfazem um determinado predicado.

80

filter

`filter` é uma função de ordem superior que filtra os elementos de uma lista que verificam um dado predicado (i.e. mantém os elementos da lista para os quais o predicado é verdadeiro).

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

Note que `(filter p lista)` é equivalente a `[x | x <- lista , p x]`

Podemos definir as funções do slide anterior à custa da função `filter`, fazendo:

```
aprov xs = filter (10<=) xs
```

```
filtraDigitos s = filter isDigit s
```

```
primQuad ps = filter aux ps
  where aux (x,y) = 0<x && 0<y
```

Ou então,

```
aprov = filter (10<=)
```

```
filtraDigitos = filter isDigit
```

```
primQuad = filter aux
  where aux (x,y) = 0<x && 0<y
```

81

Funções anónimas

Em Haskell, é possível definir novas funções através de *abstrações lambda* (λ) da forma:

`\x -> e` representando uma função com argumento formal `x` e corpo da função `e` (a notação é inspirada no λ -calculus aonde isto se escreve $\lambda x.e$)

Exemplos:

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

```
> (\y -> y*3) 4
12
```

```
> (\x -> x:x^2:x^3:[]) 2
[2,4,8]
```

Funções com mais do que um argumento podem ser definidas de forma *abreviada* por:

`\p1 ... pn -> e`

Além disso, os argumentos `p1 ... pn` podem ser *padrões*.

Exemplos:

```
> (\x y -> x+y) 5 3
8
```

```
> (\(x:xs) y -> y:xs) [3,4,5,2] 7
[7,4,5,2]
```

```
> (\(x1,y1) (x2,y2) -> (x1*x2,y1*y2)) (0,3) (5,2)
(0,6)
```

Note que: `\x y -> x+y` \equiv `\x -> (\y -> x+y)` *Justifique com base no tipo.*

Como ao definir estas funções não lhes associamos um nome, elas dizem-se **anónimas**.

82

Funções anónimas

É possível utilizar funções anónimas na definição de outras funções.

Exemplos:

```
dobro = \x->x+x
```

```
cauda = \(_:xs) -> xs
```

```
> dobro 5
10
> cauda [9,3,4,5]
[3,4,5]
```

As funções anónimas são úteis para evitar a declaração de funções auxiliares.

Exemplos:

```
trocaPares xs = map troca xs
  where troca (x,y) = (y,x)
```

```
trocaPares xs = map (\(x,y)->(y,x)) xs
```

```
primQuad = filter (\(x,y) -> 0<x && 0<y)
```

Os operadores infixos aplicados apenas a um argumento são uma forma abreviada de escrever funções anónimas.

Exemplos: $(+y) \equiv \backslash x \rightarrow x+y$

$(x+) \equiv \backslash y \rightarrow x+y$

$(*5) \equiv \backslash x \rightarrow x*5$

83

foldr

Considere as seguintes funções:

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

$\text{sum } [3,5,8] \Rightarrow 3 + (5 + (8+0))$

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

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

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

Todas estas funções têm um *padrão de computação* comum:

aplicar um operador binário ao primeiro elemento da lista e ao resultado de aplicar a função ao resto da lista.

O que se está a fazer é a extensão de uma operação binária a uma lista de operandos.

84

foldr

Podemos capturar este padrão de computação fornecendo à função `foldr` o operador binário e o resultado a devolver para a lista vazia.

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

Note que `(foldr f z [x1,...,xn])` é igual a `(f x1 (... (f xn z)...))`
ou seja, `(x1 `f` (x2 `f` (... (xn `f` z)...)))` (*associa à direita*)

Podemos definir as funções do slide anterior à custa da função `foldr`, fazendo:

```
sum xs = foldr (+) 0 xs
```

```
product xs = foldr (*) 1 xs
```

```
and bs = foldr (&&) True bs
```

```
concat ls = foldr (++) [] ls
```

Exemplos:

```
(product [4,3,5]) => 4 * (3 * (5 * 1)) => 60
```

```
(concat [[3,4,5],[2,1],[7,8]]) => [3,4,5] ++ ([2,1] ++ ([7,8]++[]))
=> [3,4,5,2,1,7,8]
```

85

foldl

Podemos usar um padrão de computação semelhante ao do `foldr`, mas *associando à esquerda*, através da função `foldl`.

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

Note que `(foldl f z [x1,...,xn])` é igual a `(f (...(f z x1) ...) xn)`
ou seja, `((...((z `f` x1) `f` x2)...)) `f` xn` (*associa à esquerda*)

Exemplos:

```
sum xs = foldl (+) 0 xs
```

```
concat ls = foldl (++) [] ls
```

```
reverse xs = foldl (\t h -> h:t) [] xs
```

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

```
concat [[2,3],[8,4,7],[1]] => ([[]]++[2,3]) ++ [8,4,7] ++ [1]
=> [2,3,8,4,7,1]
```

```
reverse [3,4] => ((\t h -> h:t) ((\t h -> h:t) [] 3) 4)
=> 4 : ((\t h -> h:t) [] 3) => 4:3:[] => [4,3]
```

86

foldr vs foldl

Note que `(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`

As funções `foldr` e `foldl` estão formemente relacionadas com as estratégias para contruir funções recursivas sobre listas que vimos atrás.

`foldr` está relacionada com a *recursividade primitiva*.

`foldl` está relacionada com o *uso de acumuladores*.

Exercício: Considere as funções

```
sumR xs = foldr (+) 0 xs
sumL xs = foldl (+) 0 xs
```

Escreva a cadeia de redução das expressões `(sumR [1,2,3])` e `(sumL [1,2,3])` e compare com o funcionamento da função somatório definida sem e com e acumuladores.

87

Outras funções de ordem superior

Composição de funções

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

Trocar a ordem dos argumentos

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

Obter a versão *curried* de uma função

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

Obter a versão *uncurried* de uma função

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

Exemplos:

```
sextuplo = dobro . triplo
```

```
reverse xs = foldl (flip (:)) [] xs
```

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

`sextuplo 5 ⇒ dobro (triplo 5) ⇒ dobro 15 ⇒ 30`

`quocientes [(3,4),(23,5),(7,3)] ⇒ [0,4,2]`

88

Novos Tipos de Dados

Para além dos *tipos básicos*, dos *tipos compostos* e dos *tipos sinónimos*, o Haskell dá ainda a possibilidade de definir **novos tipos de dados**, através de declarações da forma:

Nome do *construtor do tipo*, iniciado por letra maiúscula.

Parâmetros (*variáveis de tipo*).

```
data Nome p1 ... pn = Construtor t1 ... tm | ...  
...  
| Construtor t1 ... tk
```

Nome dos *construtores de dados* do tipo (ou simplesmente, construtores). Iniciado por letra maiúscula.

Tipo dos argumentos de cada construtor (*as variáveis de tipo dos parâmetros podem ocorrer neste tipos*).

Estas declarações definem **tipos algébricos**, eventualmente, *polimórficos*.

Cada *construtor de dados* funciona como uma função (eventualmente, constante) que recebe argumentos (do tipo indicado para o construtor) e *constroi* um valor do novo tipo de dados.

89

Tipos Algébricos

Exemplos: `data Cor = Azul | Amarelo | Verde | Vermelho`

O tipo **Cor** está a ser definido à custa de 4 construtores constantes: **Azul**, **Amarelo**, **Verde** e **Vermelho**, que serão os únicos valores deste tipo.

```
Azul :: Cor      Amarelo :: Cor  
Verde :: Cor     Vermelho :: Cor
```

A este género de tipo algébrico dá-se o nome de **tipo enumerado**.

O tipo **Bool** já pré-definido é também um exemplo de um tipo enumerado.

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

Podemos agora definir funções envolvendo estes tipos algébricos:

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

```
quente :: Cor -> Bool  
quente Amarelo = True  
quente Vermelho = True  
quente _ = False
```

90

Tipos Algébricos

Exemplo:

```
data CCart = Coord Float Float
```

Os valores do tipo **CCart** são expressões da forma $(\text{Coord } x \ y)$, em que x e y são valores do tipo **Float**.

Coord pode ser vista como uma função cujo tipo é

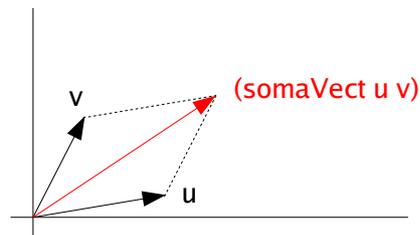
```
Coord :: Float -> Float -> CCart
```

mas os *constructores são funções especiais*, pois não têm nenhuma definição associada.

Expressões como $(\text{Coord } 1 \ 3.1)$ ou $(\text{Coord } 3 \ 0.7)$, não podem ser reduzidas, e são exemplos de valores atômicos do tipo **CCart**.

Exemplo:

Função que soma de dois vectores:



```
somaVect :: CCart -> CCart -> CCart  
somaVect (Coord x1 y1) (Coord x2 y2) = Coord (x1+x2) (y1+y2)
```

91

Tipos Algébricos

Exemplo:

```
data Hora = AM Int Int  
          | PM Int Int
```

Os valores do tipo **Hora** são expressões da forma $(\text{AM } x \ y)$ ou $(\text{PM } x \ y)$, em que x e y são valores do tipo **Int**.

Os constructores do tipo **Hora** são:

```
AM :: Int -> Int -> Hora  
PM :: Int -> Int -> Hora
```

e podem ser vistos como uma “etiqueta” que indica de que forma os argumentos a que são aplicados devem ser entendidos.

Os *data types* implementam o **co-produto** (ou a **união disjunta**) de tipos.

NOTA: Erradamente, pode parecer que termos como $(\text{AM } 5 \ 10)$, $(\text{PM } 5 \ 10)$ ou $(5, 10)$ contêm a mesma informação, mas não! Os constructores **AM** e **PM** têm aqui um papel essencial na interpretação que fazemos destes termos.

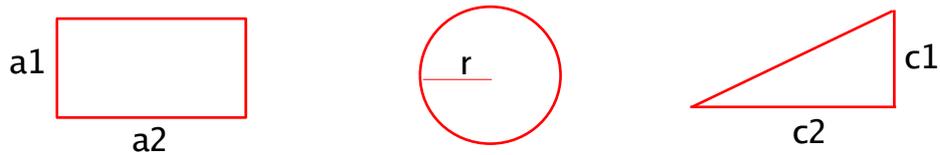
Exemplo: As funções sobre tipos algébricos geralmente definiem-se por *pattern matching*.

```
totalMinutos :: Hora -> Int  
totalMinutos (AM h m) = h*60 + m  
totalMinutos (PM h m) = (h+12)*60 + m
```

92

Tipos Algébricos

Exemplo: Um tipo de dados para representar as seguintes figuras geométricas.



```
data Figura = Rectangulo Float Float
            | Circulo Float
            | Triangulo Float Float
```

Cálculo da área de uma figura:

```
area :: Figura -> Float
area (Rectangulo a1 a2) = a1 * a2
area (Circulo r) = pi * r^2
area (Triangulo c1 c2) = c2 * c1 / 2
```

Uma lista com figuras geométricas:

```
lfig = [(Rectangulo 5 3.2), (Circulo 5.7), (Triangulo 4 3)]
```

Note que é o facto de termos definido o tipo de dados `Figura` que nos permite construir esta lista, uma vez que só são aceites *listas homogéneas*.

93

Tipos Algébricos

As definições de tipos também podem ser *recursivas*.

Exemplo: O tipo dos números naturais pode ser definido por

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

O tipo `Nat` é definido à custa dos construtores

isto é,

`Zero` é um valor do tipo `Nat`, e

se `n` é um valor do tipo `Nat`, `(Suc n)` é também um valor do tipo `Nat`.

```
Zero :: Nat
Suc  :: Nat -> Nat
```

A este género de tipo algébrico dá-se o nome de **tipo recursivo**.

Exemplos:

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

São números naturais.

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

```
somaNat :: Nat -> Nat -> Nat
somaNat Zero n = n
somaNat (Suc n) m = Suc (somaNat n m)
```

94

Tipos Algébricos

O tipo pré-definido `[a]` das listas é um outro exemplo de um *tipo recursivo*.

Exemplo: Poderíamos definir o tipo das listas, através da seguinte definição:

```
data Lista a = Nil
              | Cons a (Lista a)
```

O tipo `(Lista a)` é aqui definido à custa dos construtores

```
Nil :: Lista a
Cons :: a -> Lista a -> Lista a
```

A lista `[3, 7, 1]` seria representada pela expressão

```
Cons 3 (Cons 7 (Cons 1 Nil))
```

`(Lista a)` é um exemplo de um **tipo polimórfico**.

`Lista` está parameterizada com uma variável de tipo `a`, que poderá ser substituída por um tipo qualquer. (É neste sentido que se diz que `Lista` é um *construtor de tipos*.)

Exemplo:

```
comprimento :: Lista a -> Int
comprimento Nil = 0
comprimento (Cons _ xs) = 1 + comprimento xs
```

95

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:

de um dado tipo `T` de valores do tipo `T`

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

todas de um mesmo tipo

Exemplos:

```
fria :: Cor -> Bool
fria c = case c of Azul -> True
                  Verde -> True
                  _ -> False
```

```
comprimento :: Lista a -> Int
comprimento l = case l of
  Nil -> 0
  (Cons _ xs) -> 1 + comprimento xs
```

96

Expressões case

Exemplos:

```
par :: Nat -> Bool
par n = case n of
  Zero          -> True
  (Suc (Suc x)) -> par x
  -             -> False
```

```
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p l = case l of
  []      -> []
  (x:xs) -> if (p x) then x : (takeWhile p xs)
              else []
```

Exercícios:

- Defina duas versões da função `impar` (com e sem expressões case).
- Defina uma outra versão da função `takeWhile` utilizando várias equações.

Nota: As expressões `if-then-else` são equivalentes à análise de casos no tipo `Bool`.

```
if e then e1
   else e2      é equivalente a      case e of True -> e1
                                         False -> e2
```

97

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);
- tipos recursivos;
- uma certa forma de *encapsulamento de dados*.

Além disso, os tipos algébricos:

(falaremos destes aspectos mais tarde)

- podem ter uma apresentação escrita própria
- podem ser declarados como instâncias de classes

Nota: Se quiser experimentar os exemplos apresentados atrás, será melhor acrescentar às declarações dos tipos algébricos a indicação: **deriving Show**, para que os valores dos novos tipos possam ser escritos (no formato usual).

Exemplo:

```
data Nat = Zero | Suc Nat
         deriving Show
```

98

O construtor de tipos `Maybe`

Um tipo algébrico importante, já pré-definido no `Prelude` é o tipo polimórfico

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

que permite representar a *parcialidade*, podendo ser usado para lidar com situações de exceções e erros.

Exemplos:

```
divisao :: Integer -> Integer -> Maybe Integer
divisao x y | y /= 0 = Just (x `div` y)
            | otherwise = Nothing
```

```
cabeca :: [a] -> Maybe a
cabeca (x:xs) = Just x
cabeca [] = Nothing
```

em casos de exceções o resultado é `Nothing`

Funções que trabalham sobre um tipo `t` terão que ser adaptadas para trabalhar com o tipo `Maybe t`.

Exemplo:

```
soma (Just x) (Just y) = Just (x+y)
soma _ _ = Nothing
```

99

Exemplo: A seguinte função que procura o nome associado a um dado número de BI, numa tabela implementada como uma lista de pares.

```
type BI = Integer
type Nome = String
```

```
procura :: BI -> [(BI, Nome)] -> Nome
procura n ((x,y):xys) | n == x      = y
                    | otherwise = procura n xys
procura _ [] = error "Não existe!"
```

A função `procura` termina em erro caso o BI não exista na tabela. Ou seja, `procura` é uma *função parcial*.

Podemos *totalizar* a função de procura usando o tipo `(Maybe Nome)`.

```
proc :: BI -> [(BI, Nome)] -> Maybe Nome
proc n ((x,y):xys) | n == x      = Just y
                  | otherwise = proc n xys
proc _ [] = Nothing
```

Desta forma, se o BI não existir na tabela a função `proc` devolve `Nothing` e nunca termina em erro. Ou seja, `proc` é uma *função total*.

100

A função `proc` só consegue concluir que um dado BI não ocorre na tabela, ao fim de pesquisar toda a lista.

Esta conclusão poderia ser tirada mais cedo, se que a lista estivesse *ordenada* por BI.

Exemplo: Tendo a garantia de que a lista está ordenada por ordem crescente de BI, podemos definir a função de procura da seguinte forma:

```
proc :: BI -> [(BI, Nome)] -> Maybe Nome
proc n ((x,y):xys) | n > x = proc n xys
                  | n == x = Just y
                  | n < x = Nothing
proc _ [] = Nothing
```

Nos casos de insucesso, esta versão de `proc` é bastante *mais eficiente* do que a versão do slide anterior.

Exercício: Compare o funcionamento das duas versões da função `proc` para o seguinte exemplo:

```
proc 3 [ (bi, "xxxxx") | bi <- [1,5..1000] ]
```

101

Árvores Binárias

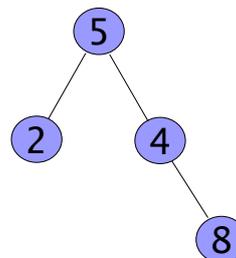
Uma estrutura de dados muito útil para organizar informação são as *árvores binárias*.

O tipo polimórfico das árvores binárias pode definido pelo seguinte tipo recursivo:

```
data ArvBin a = Vazia
              | Nodo a (ArvBin a) (ArvBin a)
```

Ou seja, uma árvore binária: ou é vazia; ou é um nodo com um valor e duas sub-árvores.

Exemplo: A árvore de valores inteiros:



é representada pela expressão

```
(Nodo 5 (Nodo 2 Vazia Vazia)
        (Nodo 4 Vazia (Nodo 8 Vazia Vazia))
)
```

de tipo `(ArvBin Int)`.

102

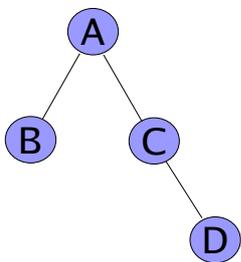
As funções definidas sobre tipos de dados recursivos, são geralmente funções recursivas, com *padrões de recursividade semelhantes aos dos tipos de dados*.

Exemplo:

```
somaL :: [Int] -> Int
somaL [] = 0
somaL (x:xs) = x + (somaL xs)
```

```
somaA :: ArvBin Int -> Int
somaA Vazia = 0
somaA (Nodo x esq dir) = x + (somaA esq) + (somaA dir)
```

Terminologia



O nó A é a **raiz** da árvore
Os nós B e C são **filhos** (ou **descendentes**) de A
O nó C é **pai** de D

O **caminho** (*path*) de um nó é a sequência de nós da raiz até esse nó.

A **altura** é o comprimento do caminho mais longo.

103

Funções sobre árvores binárias

Exemplos:

```
altura :: ArvBin a -> Integer
altura Vazia = 0
altura (Nodo _ e d) = 1 + max (altura e) (altura d)
```

```
mapAB :: (a -> b) -> ArvBin a -> ArvBin b
mapAB f Vazia = Vazia
mapAB f (Nodo x e d) = Nodo (f x) (mapAB f e) (mapAB f d)
```

```
unzipAB :: ArvBin (a,b) -> (ArvBin a, ArvBin b)
unzipAB Vazia = (Vazia, Vazia)
unzipAB (Nodo (x,y) e d) = let (e1,e2) = unzipAB e
                               (d1,d2) = unzipAB d
                               in (Nodo x e1 d1, Nodo y e2 d2)
```

Exercício: Defina as funções

contaNodos :: ArvBin a -> Integer

zipAB :: ArvBin a -> ArvBin b -> ArvBin (a,b)

104

Travessias de árvores binárias

Para converter uma árvore binária numa lista podemos usar diversas estratégias, como por exemplo:

Preorder: R E D R – visitar a raiz
Inorder: E R D E – atravessar a sub-árvore esquerda
Postorder: E D R D – atravessar a sub-árvore direita

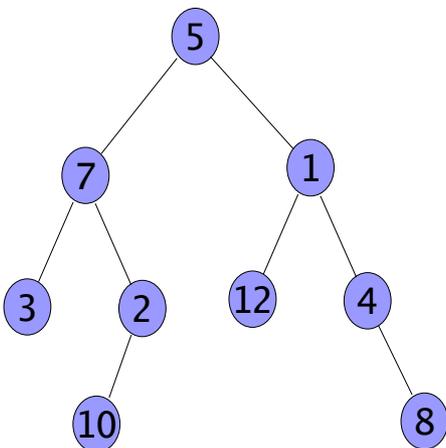
```
preorder :: ArvBin a -> [a]
preorder Vazia = []
preorder (Node x e d) = [x] ++ (preorder e) ++ (preorder d)
```

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

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

105

```
arv = (Nodo 5 (Nodo 7 (Nodo 3 Vazia Vazia)
                  (Nodo 2 (Nodo 10 Vazia Vazia) Vazia)
              )
      (Nodo 1 (Nodo 12 Vazia Vazia)
              (Nodo 4 Vazia (Nodo 8 Vazia Vazia))
          )
  )
```



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]

106

Árvores Binárias de Procura

Uma **árvore binária** diz-se de **procura**, se é vazia, ou se verifica todas as seguintes condições:

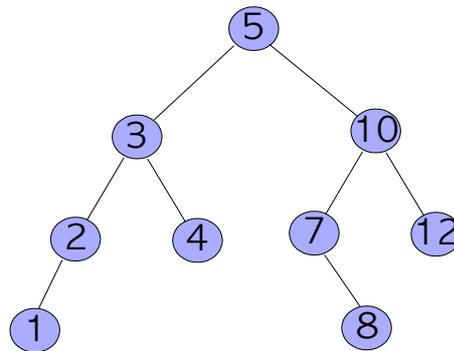
- a raiz da árvore é maior do que todos os elementos da sub-árvore esquerda;
- a raiz da árvore é menor do que todos os elementos da sub-árvore direita;
- ambas as sub-árvores são árvores binárias de procura.

Exemplo: O seguinte predicado para testar se uma dada árvore binária é de procura, está errado. Porquê?

```
arvBinProcura Vazia = True
arvBinProcura (Nodo x e d) =
  (x > maximum (preorder e)) && (x < minimum (preorder d))
  && (arvBinProcura e) && (arvBinProcura d)
```

Exemplo: A árvore seguinte é uma árvore binária de procura.

Qual é o termo que a representa ?



111

Exemplo: Acrescentar um elemento à árvore binária de procura.

```
insABProc x Vazia = (Nodo x Vazia Vazia)
insABProc x (Nodo y e d)
  | x < y = Nodo y (insABProc x e) d
  | y < x = Nodo y e (insABProc x d)
  | x == y = Nodo y e d
```

Note que os elementos repetidos não estão a ser acrescentados à árvore de procura.

O que alteraria para, relaxando a noção de árvore binária de procura, aceitar elementos repetidos na árvore ?

Exercício: Qual é a função de travessia que aplicada a uma árvore binária de procura retorna uma lista ordenada com os elementos da árvore ?

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

Exercício: Desenhe as árvores resultantes das seguintes sequências de inserção numa árvore inicialmente vazia.

- 7, 4, 9, 6, 1, 8, 5
- 1, 4, 5, 6, 7, 8, 9
- 6, 4, 1, 8, 9, 5, 7

Exercício: Defina uma função que recebe uma lista e constroi uma árvore binária de procura com os elementos da lista.

Árvores Balanceadas

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

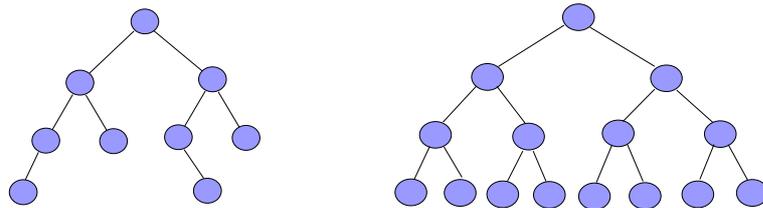
- as alturas da sub-árvores esquerda e direita diferem no máximo em uma unidade;
- ambas as sub-árvores são árvores balanceadas.

Exemplo: Predicado para testar se uma dada árvore binária é balanceada.

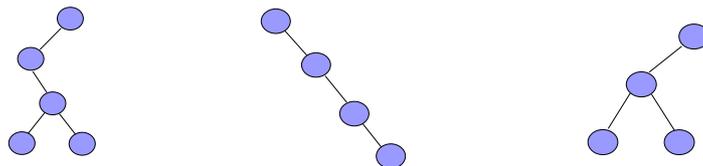
```
balanceada Vazia = True
balanceada (Nodo _ e d) = (abs ((altura e)-(altura d))) <= 1
                        && (balanceada e) && (balanceada d)
```

Exemplos:

Balanceadas:



Não balanceadas:



109

As árvores binárias de procura são estruturas de dados que possibilitam **pesquisas potencialmente mais eficientes** da informação, do que as pesquisas em listas.

Exemplo:

A tabela de associações BI – Nome, pode ser guardada numa árvore binária de procura com o tipo **ArvBin (BI, Nome)**.

A função de pesquisa nesta árvores binária de procura organizada por BI pode ser definida por

```
pesquisaABProc :: BI -> ArvBin (BI, Nome) -> Maybe Nome
pesquisaABProc n Vazia = Nothing
pesquisaABProc n (Nodo (x,y) e d)
    | n == x = Just y
    | n < x  = pesquisaABProc n e
    | n > x  = pesquisaABProc n d
```

110

Chama-se **chave** ao componente de informação que é *único* para cada entidade. Por exemplo: o nº de BI é chave para cada cidadão; nº de aluno é chave para cada estudante universitário; nº de contribuinte é chave para cada empresa.

Uma medida da **eficiência** de uma pesquisa é o número de comparações de chaves que são feitas até que se encontre o elemento a pesquisar. É claro que isso depende da posição da chave na estrutura de dados.

O número de comparações de chaves numa pesquisa:

- *numa lista*, é no máximo igual ao comprimento da lista;
- *numa árvore binária de procura*, é no máximo igual à altura da árvore.

Assim, a pesquisa em árvores binárias de procura são especialmente mais eficientes se as árvores forem balanceadas.

Porquê ?

111

Existem algoritmos de inserção que mantêm o equilíbrio das árvores (mas não serão apresentados nesta disciplina).

Exemplo: A partir de uma lista ordenada por ordem crescente de chaves podemos construir uma árvore binária de procura balanceada, através da função

```
constroiArvBal [] = Vazia
constroiArvBal xs = Nodo x (constroiArvBal xs1) (constroiArvBal xs2)
  where
    k = (length xs) `div` 2
    xs1 = take k xs
    (x:xs2) = drop k xs
```

Exercícios:

- Defina uma função que dada uma árvore binária de procura, devolve o seu valor mínimo.
- Defina uma função que dada uma árvore binária de procura, devolve o seu valor máximo.
- Como poderá ser feita a remoção de um nodo de uma árvore binária de procura, de modo a que a árvore resultante continue a ser de procura ? Defina uma função que implemente a estratégia que indicou.

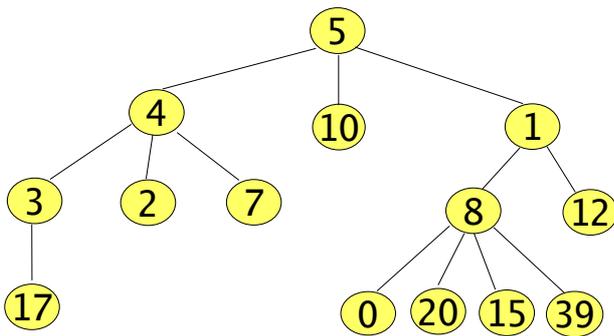
112

Outras Árvores

Árvores Irregulares

(finitely branching trees)

```
data Tree a = Node a [Tree a]
```



Esta árvore do tipo **(Tree Int)** é representada pelo termo:

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

113

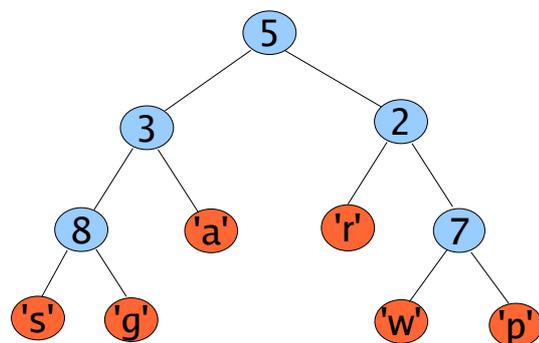
Outras Árvores

Full Trees

Árvores com *nós* (intermédios) do tipo **a** e *folhas* do tipo **b**.

```
data ABin a b = Folha b
              | No a (ABin a b) (ABin a b)
```

Esta árvore do tipo **(ABin Int Char)** é representada pelo termo:



```
(No 5 (No 3 (No 8 (Folha 's') (Folha 'g'))
           (Folha 'a'))
      )
  (No 2 (Folha 'r')
       (No 7 (Folha 'w') (Folha 'p')))
  )
)
```

114

“Records”

Numa declaração de um tipo algébrico, os construtores podem ser declarados associando a cada um dos seus parâmetros um nome (uma *etiqueta*).

Exemplo:

```
data PontoC = Pt {xx :: Float, yy :: Float, cor :: Cor}
```

desta forma, para além do construtor de dados

```
Pt :: Float -> Float -> Cor -> PontoC
```

também ficam definidos os nome dos *campos* **xx**, **yy** e **cor**, e 3 *selectores* com o mesmo nome:

```
xx :: PontoC -> Float
yy :: PontoC -> Float
cor :: PontoC -> Cor
```

Os valores do novo tipo PontoC podem ser construídos da forma usual, por aplicação do construtor aos seus argumentos.

```
p1 = (Pt 3.2 5.5 Azul) :: PontoC
```

Além disso, o nome dos campos podem agora também ser usados na construção de valores do novo tipo.

```
p2 = Pt {xx=3.1, yy=8.0, cor=Vermelho} :: PontoC
p3 = Pt {cor=Verde, yy=2.2, xx=7.1}    :: PontoC
```

115

“Records”

Note que $\left\{ \begin{array}{l} (Pt\ 3.2\ 5.5\ Azul) \\ Pt\ \{xx=3.2,\ yy=5.5,\ cor=Azul\} \\ Pt\ \{yy=5.5,\ cor=Azul,\ xx=3.2\} \end{array} \right\}$ são exactamente o mesmo valor.

Aos tipos com um único construtor e com os campos etiquetados dá-se o nome de *records*.

Os *padrões* podem também usar o nome dos campos (todos ou alguns, por qualquer ordem).

Exemplo: Três versões equivalentes da função que calcula a distância de um ponto à origem.

```
dist0 :: PontoC -> Float
dist0 p = sqrt ((xx p)^2 * (yy p)^2)
```

```
dist0' :: PontoC -> Float
dist0' Pt {xx=x, yy=y} = sqrt (x^2 * y^2)
```

```
dist0'' :: PontoC -> Float
dist0'' (Pt x y c) = sqrt (x^2 * y^2)
```

116

“Records”

Sendo **p** um valor do tipo `PontoC`, **p {xx=0}** é um novo valor com o campo `xx=0` e os restantes campos com o valor que tinham em **p**.

Exemplos:

```
p1 {cor = Amarelo} ≡ Pt {xx=3.2, yy=5.5, cor=Amarelo}
p3 {xx=0, yy=0} ≡ Pt {xx=0, yy=0, cor=Verde}
```

```
simetrico :: PontoC -> PontoC
simetrico p = p {xx=(yy p), yy=(xx p)}
```

É possível ter campos etiquetados em tipos com mais de um construtor. Um campo não pode aparecer em mais do que um tipo, mas dentro de um tipo pode aparecer associado a mais de um construtor, desde que tenha o mesmo tipo.

Exemplo:

```
data EX = C1 { s :: Int, r :: Float }
         | C2 { s :: Int, w :: String }
```

117

Polimorfismo paramétrico

Com já vimos, o sistema de tipos do Haskell incorpora **tipos polimórficos**, isto é, tipos com variáveis (*quantificadas universalmente*, de forma implícita).

Exemplos:

Para qualquer tipo **a**, `[a]` é o tipo das listas com elementos do tipo **a**.

Para qualquer tipo **a**, `(ArvBin a)` é o tipo das árvores binárias com nodos do tipo **a**.

As variáveis de tipo podem ser vistas como *parâmetros (dos constructores de tipos)* que podem ser substituídos por tipos concretos. Esta forma de polimorfismo tem o nome de **polimorfismo paramétrico**.

Exemplo:

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

length [5.6,7.1,2.0,3.8] ⇒ 4
length ['a','b','c'] ⇒ 3
length [(3,True),(7,False)] ⇒ 2

Prelude> :t length
length :: forall a. [a] -> Int
```

O tipo `[a]->Int` não é mais do que uma abreviatura de `∀a. [a]->Int` :

“para todo o tipo a, [a]->Int é o tipo das funções com domínio em [a] e contradomínio Int”.

118

Polimorfismo *ad hoc* (sobrecarga)

O Haskell incorpora ainda uma outra forma de polimorfismo que é a **sobrecarga de funções**. Um mesmo identificador de função pode ser usado para designar funções computacionalmente distintas. A esta característica também se chama **polimorfismo *ad hoc***.

Exemplos:

O operador **(+)** tem sido usado para somar, tanto valores inteiros como valores decimais.

O operador **(==)** pode ser usado para comparar inteiros, caracteres, listas de inteiros, strings, booleanos, ...

Afinal, qual é o tipo de **(+)** ? E de **(==)** ?

A sugestão `(+) :: a -> a -> a`
`(==) :: a -> a -> Bool` **não serve**, pois são tipos demasiado genéricos !

Faria com que fossem aceites expressões como, por exemplo:

`('a' + 'b')` , `(True + False)` , `("esta'" + "errado")` ou `(div == mod)` ,

e estas expressões resultariam em **erro**, pois estas operações não estão preparadas para trabalhar com valores destes tipos.

Em Haskell esta situação é resolvida através de **tipos qualificados** (*qualified types*), fazendo uso da noção de **classe**.

119

Tipos qualificados

Conceptualmente, um **tipo qualificado** pode ser visto como um tipo polimórfico só que, em vez da quantificação universal da forma *“para todo o tipo a, ...”* vai-se poder dizer *“para todo o tipo a que pertence à classe C, ...”*. Uma classe pode ser vista como um conjunto de tipos.

Exemplo:

Sendo **Num** uma classe (*a classe dos números*) que tem como elementos os tipos: `Int`, `Integer`, `Float`, `Double`, ..., pode-se dar a **(+)** o tipo preciso de:

$\forall a \in \text{Num}. a \rightarrow a \rightarrow a$

o que em Haskell se vai escrever: `(+) :: Num a => a -> a -> a`

e lê-se: *“para todo o tipo a que pertence à classe Num, (+) tem tipo a->a->a”*.

Uma classe surge assim como uma forma de classificar tipos (quanto às funcionalidades que lhe estão associadas). Neste sentido as classes podem ser vistas como os **tipos dos tipos**.

Os tipos que pertencem a uma classe também serão chamados de **instâncias** da classe.

A capacidade de **qualificar** tipos polimórficos é uma característica inovadora do Haskell.

120

Classes & Instâncias

Uma **classe** estabelece um conjunto de assinaturas de funções (os **métodos da classe**). Os tipos que são declarados como **instâncias** dessa classe têm que ter definidas essas funções.

Exemplo: A seguinte declaração (simplificada) da classe **Num**

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

impõe que todo o tipo **a** da classe **Num** tenha que ter as operações **(+)** e **(*)** definidas.

Para declarar **Int** e **Float** como elementos da classe **Num**, tem que se fazer as seguintes **declarações de instância**

```
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.

Se $x :: \text{Int}$ e $y :: \text{Int}$ então $x + y \equiv x \text{ `primPlusInt` } y$
Se $x :: \text{Float}$ e $y :: \text{Float}$ então $x + y \equiv x \text{ `primPlusFloat` } y$

121

Tipo principal

O **tipo principal** de uma expressão ou de uma função 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 ou função.

Qualquer expressão ou função válida tem um tipo principal **único**. O Haskell **infere** sempre o tipo principal das expressões ou funções, mas é sempre possível associar tipos mais específicos (que são instância do tipo principal).

Exemplo: O tipo principal inferido pelo haskell para o operador **(+)** é

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

Mas,

```
(+) :: Int -> Int -> Int
(+) :: Float -> Float -> Float
```

são também tipos válidos dado que tanto **Int** como **Float** são instâncias da classe **Num**, e portando podem substituir a variável **a**.

Note que **Num a** não é um tipo, mas antes uma restrição sobre um tipo. Diz-se que **(Num a)** é o **contexto** para o tipo apresentado.

Exemplo:

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

O tipo principal da função **sum** é

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

- $\text{sum} :: [a] \rightarrow a$ seria um tipo demasiado geral. **Porquê ?**
- **Qual será o tipo principal da função `product` ?**

122

Definições por defeito

Relembre a definição da função pré-definida `elem`:

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

Qual será o seu tipo ?

É necessário que `(==)` esteja definido para o tipo dos elementos da lista.

Existe pré-definida a classe **Eq**, dos tipos para os quais existe uma operação de igualdade.

```
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 defeito** para estes métodos (*default methods*).

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

123

Exemplos de instâncias de Eq

O tipo **Cor** é uma instância da classe **Eq** com `(==)` definido como se segue:

O método `(/=)` está definido por defeito.

```
instance Eq Cor where
  Azul == Azul      = True
  Verde == Verde    = True
  Amarelo == Amarelo = True
  Vermelho == Vermelho = True
  _ == _           = False
```

`(==)` de **Nat**

O tipo **Nat** também pode ser declarado como instância da classe **Eq**:

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

O tipo **PontoC** com instância de **Eq**:

```
instance Eq PontoC where
  (Pt x1 y1 c1) == (Pt x2 y2 c2) = (x1==x2) && (y1==y2)
  && (c1==c2)
```

`(==)` de **Float**

`(==)` de **Cor**

Nota: `(==)` é uma função recursiva em **Nat**, mas não em **PontoC**.

124

Instâncias com restrições

Relembre a definição das árvores binárias.

```
data ArvBin a = Vazia
              | Nodo a (ArvBin a) (ArvBin a)
```

Como poderemos fazer o teste de igualdade para árvores binárias ?

Duas árvores são iguais se tiverem a mesma estrutura (a mesma forma) e se os valores que estão nos nodos também forem iguais.

Portanto, para fazer o teste de igualdade em `(ArvBin a)`, necessariamente, tem que se saber como testar a igualdade entre os valores que estão nos nodos, i.e., em `a`.

Só poderemos declarar `(ArvBin a)` como instância da classe `Eq` se `a` for também uma instância da classe `Eq`.

Este tipo de *restrição* pode ser colocado na declaração de instância, fazendo:

```
instance (Eq a) => Eq (ArvBin a) where
  Vazia == Vazia = True
  (Nodo x1 e1 d1) == (Nodo x2 e2 d2) = (x1==x2) && (e1==e2)
                                     && (d1==d2)
  _ == _ = False
```

(==) de `a` (==) de `(ArvBin a)`

125

Instâncias derivadas de Eq

O testes de igualdade definidos até aqui implementam a **igualdade estrutural** (*dois valores são iguais quando resultam do mesmo construtor aplicado a argumentos também iguais*).

Quando assim é pode-se evitar a declaração de instância se na declaração do tipo for acrescentada a instrução **deriving Eq**.

Exemplos: Com esta declarações, o Haskell deriva automaticamente declarações de instância de `Eq` (iguais às que foram feitas) para estes tipos.

```
data Cor = Azul | Amarelo | Verde | Vermelho
         deriving Eq
```

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

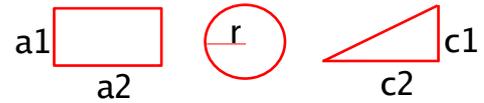
```
data PontoC = Pt {xx :: Float, yy :: Float, cor :: Cor}
            deriving Eq
```

```
data ArvBin a = Vazia
              | Nodo a (ArvBin a) (ArvBin a)
            deriving Eq
```

Mas, nem sempre a igualdade estrutural é a desejada.

Exemplo: Relembre o tipo de dados [Figura](#):

```
data Figura = Rectangulo Float Float
            | Circulo Float
            | Triangulo Float Float
```



Neste caso queremos que duas figuras sejam consideradas iguais ainda que a ordem pela qual os valores são passados possa ser diferente.

```
instance Eq Figura where
    (Rectangulo x1 y1) == (Rectangulo x2 y2) =
        ((x1==x2) && (y1==y2)) || ((x1==y2) && (x2==y1))
    (Circulo r1) == (Circulo r2) = r1==r2
    (Triangulo x1 y1) == (Triangulo x2 y2) =
        ((x1==x2) && (y1==y2)) || ((x1==y2) && (x2==y1))
```

127

Exercícios:

- Considere a seguinte definição de tipo, para representar horas nos dois formatos usuais.

```
data Time = Am Int Int
          | Pm Int Int
          | Total Int Int
```

Declare [Time](#) como instância da classe [Eq](#) de forma a que (`==`) teste se dois valores representam a mesma hora do dia, independentemente do seu formato.

- Qual o tipo principal da seguinte função:

```
lookup x ((y,z):yzs) | x /= y = lookup x yzs
                  | otherwise = Just z
lookup _ [] = Nothing
```

- Considere a seguinte declaração: `type Assoc a b = [(a,b)]`

Será que podemos declarar (`Assoc a b`) como instância da classe `Eq` ?

128

Herança

O sistema de classes do Haskell também suporta a noção de **herança**.

Exemplo: Podemos definir a classe `Ord` como uma **extensão** da classe `Eq`.

-- isto é uma simplificação da classe `Ord` já pré-definida

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

A classe `Ord` **herda** todos os métodos de `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 necessariamente que ser instância de `Eq`.

Exemplo:

```
estaABProc :: Ord a => a -> ArvBin a -> Bool
estaABProc _ Vazia = False
estaABProc x (Nodo y e d) | x < y = estaABProc x e
                          | x > y = estaABProc x d
                          | x == y = True
```

A restrição `(Eq a)` não é necessária. **Porquê ?**

129

Herança múltipla

O sistema de classes do Haskell também suporta **herança múltipla**. Isto é, uma classe pode ter mais do que uma superclasse.

Exemplo: A classe `Real`, já pré-definida, tem a seguinte declaração

```
class (Num a, Ord a) => Real a where
  toRational :: a -> Rational
```

A classe `Real` herda todos os métodos da classe `Num` e da classe `Ord` e estabelece mais uma função.

NOTA: Na declaração dos tipos dos métodos de uma classe, é possível colocar restrições às variáveis de tipo, excepto à variável de tipo da classe que está a ser definida.

Exemplo:

```
class C a where
  m1 :: Eq b => (b,b) -> a -> a
  m2 :: Ord b => a -> b -> b -> a
```

O método `m1` impõe que `b` pertença à classe `Eq`, e o método `m2` impõe que `b` pertença a `Ord`. Restrições à variável `a`, se forem necessárias, terão que ser feitas no contexto da classe, e nunca ao nível dos métodos.

130

A classe Ord

```
data Ordering = LT | EQ | GT
              deriving (Eq, Ord, Ix, Enum, Read, Show, Bounded)
```

```
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
```

131

Exemplos de instâncias de Ord

Exemplo:

```
instance Ord Nat where
  compare (Suc _) Zero = GT
  compare Zero (Suc _) = LT
  compare Zero Zero    = EQ
  compare (Suc n) (Suc m) = compare n m
```

Instâncias da classe `Ord` podem ser **derivadas automaticamente**. Neste caso, a relação de ordem é estabelecida com base na ordem em que os construtores são apresentados e na relação de ordem entre os parâmetros dos construtores.

Exemplo:

```
data AB a = V | NO a (AB a) (AB a)
          deriving (Eq, Ord)
```

```
ar1 = NO 1 V V
ar2 = NO 2 V V
```

Será que poderíamos não derivar Eq ?

```
> V < ar1
True
> ar1 < ar2
True
> (NO 4 ar1 ar2) < (NO 5 ar2 ar1)
True
> (NO 4 ar1 ar2) < (NO 3 ar2 ar1)
False
> (NO 4 ar1 ar2) < (NO 4 ar2 ar1)
True
```

132

As restrições às variáveis de tipo que são impostas pelo contexto, *propagam-se* ao logo do processo de inferência de tipos do Haskell.

Exemplo: Relembre a definição da função `quicksort`.

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

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) = let (l1,l2) = parte x xs
                    in (quicksort l1)++[x]++(quicksort l2)
```

Note como o contexto `(Ord a)` do tipo da função `parte` se propaga para a função `quicksort`.

133

A classe Show

A classe `Show` estabelece métodos para converter um valor de um tipo qualquer (que lhe pertença) numa string.

O interpretador Haskell usa o método `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
```

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

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

A função `showsPrec` usa uma string como acumulador. É muito eficiente.

134

Exemplos de instâncias de Show

Exemplo:

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

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

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

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.

Exemplo: Se, em alternativa, tivéssemos feito

```
data Nat = Zero | Suc Nat
  deriving Show
```

teríamos

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

Exemplo:

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

```
> (AM 9 30)
9:30 am
```

```
> (PM 1 35)
1:35 pm
```

135

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.

```
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**.

```
Prelude> :t 35
35 :: Num a => a
```

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

```
Prelude> 35 + 2.1
37.1
```

136

Exemplos de instâncias de Num

Exemplo:

```
instance Num Nat where
  (+) = somaNat
  (*) = prodNat
  (-) = subNat
  fromInteger = deInteger
  abs = id
  signum = sinal
  negate n = error "indefinido ..."
```

Note que `Nat` já pertence às classes `Eq` e `Show`.

```
prodNat :: Nat -> Nat -> Nat
prodNat Zero _ = Zero
prodNat (Suc n) m = somaNat m (prodNat n m)
```

```
subtNat :: Nat -> Nat -> Nat
subtNat n Zero = n
subtNat (Suc n) (Suc m) = subtNat n m
subtNat Zero _ = error "indefinido ..."
```

```
sinal :: Nat -> Nat
sinal Zero = Zero
sinal (Suc _) = Suc Zero
```

```
deInteger :: Integer -> Nat
deInteger 0 = Zero
deInteger (n+1) = Suc (deInteger n)
deInteger _ = error "indefinido ..."
```

```
somaNat :: Nat -> Nat -> Nat
somaNat Zero n = n
somaNat (Suc n) m = Suc (somaNat n m)
```

137

```
tres = Suc (Suc (Suc Zero))
quatro = Suc tres
```

usa o método
`show`

```
> tres + quatro
7
```

método da classe `Num`
`somaNat`

```
> tres * quatro
12
```

método da classe `Num`
`prodNat`

```
> tres + 10
13
```

Nota: Não é possível derivar automaticamente instâncias da classe `Num`.

138

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`, `Char`, `Bool`, ...

Exemplos:

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

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

139

Exemplos de instâncias de Enum

Exemplo:

```
instance Enum Nat where
  toEnum = intToNat
  where intToNat :: Int -> Nat
        intToNat 0 = Zero
        intToNat (n+1) = Suc (intToNat n)

  fromEnum = natToInt
```

```
> [Zero, tres .. (tres * tres)]
[0,3,6,9]
> [Zero .. tres]
[0,1,2,3]
> [(Suc Zero), tres ..]
[1,3,5,7,9,11,13,15,17,19,21,23,25, ...]
```

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

Exemplo:

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

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

140

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`).

```
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"
```

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

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

`lex` é um *analisador léxico* definido no `Prelude`.

141

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.*)

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 (Show,Read)
```

```
data Nat = Zero | Suc Nat
  deriving Read
```

```
> read "Am 8 30" :: Time
Am 8 30
> read "(Total 17 15)" :: Time
Total 17 15
> read "Suc (Suc Zero)" :: Nat
2
> read "[2,3,6,7]" :: [Int]
[2,3,6,7]
> read "[Zero, Suc Zero]" :: [Nat]
[0,1]
```

É necessário indicar o tipo do valor a produzir.

Quase todos os tipos pré-definidos pertencem à classe `Read`.

Porquê?

142

Declaração de tipos polimórficos com restrições nos parâmetros

Na declaração de um **tipo algébrico** pode-se exigir que os parâmetros pertençam a determinadas classes.

Exemplo:

```
data (Ord a) => STree a = Null
    | Branch a (STree a) (STree a)
```

```
delSTree x Null = Null
delSTree x (Branch y e Null) | x == y = e
delSTree x (Branch y Null d) | x == y = d
delSTree x (Branch y e d)
    | x < y = Branch y (delSTree x e) d
    | x > y = Branch y e (delSTree x d)
    | x == y = let z = minSTree d
                in Branch z e (delSTree z d)
```

```
minSTree (Branch x Null _) = x
minSTree (Branch _ e _) = minSTree e
```

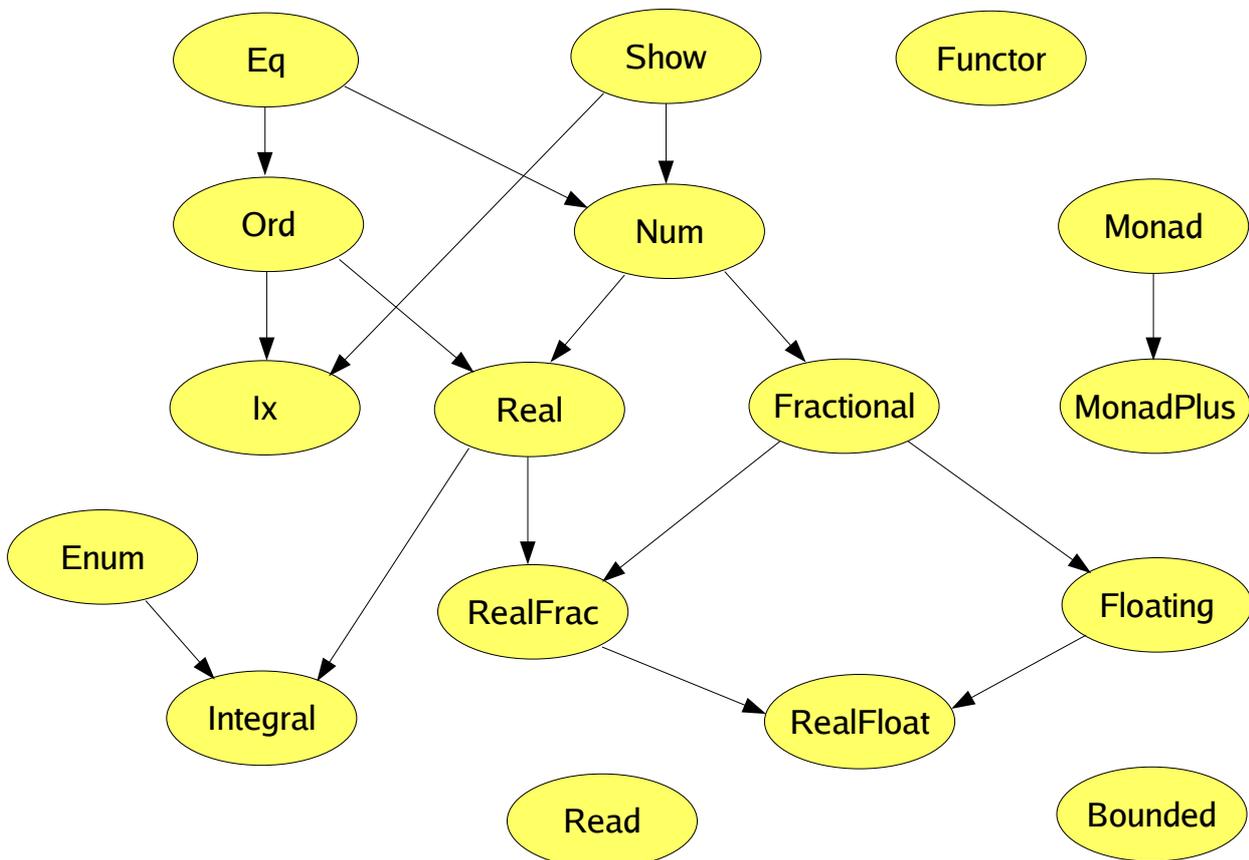
Na declaração de **tipos sinónimos** também se podem impôr restrições de classes.

Exemplo:

```
type TAssoc a b = (Eq a) => [(a,b)]
```

143

Hierarquia de classes pré-definidas do Haskell



```
Prelude> :i Nome_da_Classe
```

144

Classes de Construtores de Tipos

Relembre os tipos paramétricos (`Maybe a`), `[a]`, (`ArvBin a`), (`Tree a`) ou (`ABin a b`).

`Maybe`, `[]`, `ArvBin`, `Tree` e `ABin`, não são tipos, mas podem ser vistos como operadores sobre tipos – são **construtores de tipos**.

Exemplo: `Maybe` não é um tipo, mas (`Maybe Int`) é um tipo que resulta de aplicar o construtor de tipos `Maybe` ao tipo `Int`.

Em Haskell é possível definir classes de construtores de tipos. Um exemplo disso é a classe `Functor`:

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

Note que `f` não é um tipo.
`f a` e `f b` é que são tipos.

Exemplos:

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

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

Note que o que se está a declarar como instância da classe `Functor` são construtores de tipos.

```
instance Functor ArvBin where
  fmap = mapAB
```

145

Definição de novas classes

Para além da hierarquia de classes pré-definidas, o Haskell permite **definir novas classes**.

Exemplo: Podemos definir a classe das *ordens parciais* da seguinte forma

```
class (Eq a) => OrdParcial a where
  comp :: a -> a -> Maybe Ordering      -- basta definir comp

  lt, gt, eq :: a -> a -> Maybe Bool
  lt x y = case (comp x y)
    of { Nothing -> Nothing ; (Just LT) -> Just True ; _ -> Just False }
  gt x y = case (comp x y)
    of { Nothing -> Nothing ; (Just GT) -> Just True ; _ -> Just False }
  eq x y = case (comp x y)
    of { Nothing -> Nothing ; (Just EQ) -> Just True ; _ -> Just False }

  maxi, mini :: a -> a -> Maybe a
  maxi x y = case (comp x y) of
    Nothing -> Nothing
    Just GT -> Just x
    _       -> Just y
  mini x y = case (comp x y) of
    Nothing -> Nothing
    Just LT -> Just x
    _       -> Just y
```

Nota: Repare nos diversos modos de escrever expressões case.

146

A relação de *inclusão de conjuntos* é um bom exemplo de uma relação de ordem parcial.

Exemplo: A noção de conjunto pode ser implementada pelo tipo

```
data (Eq a) => Conj a = C [a] deriving Show
```

É necessário que se consiga fazer o teste de pertença.

```
instance (Eq a) => OrdParcial (Conj a) where
  comp (C u) (C v) = let p1 = u `contido` v
                      p2 = v `contido` u
                      in if p1 && p2 then Just EQ else
                          if p1      then Just LT else
                          if p2      then Just GT
                          else Nothing
  where
    contido :: (Eq a) => [a] -> [a] -> Bool
    contido xs ys = all (\x-> elem x ys) xs
```

```
> (C [2,1]) `gt` (C [7,1,5,2])
Just False
> (C [2,1,3]) `lt` (C [7,1,5])
Nothing
```

```
> (C [2,1,2,1]) `lt` (C [7,1,5,5,2])
Just True
> (C [3,3,5,1]) `eq` (C [5,1,5,3,1])
Just True
```

147

A noção de *função finita* estabelece um conjunto de associações entre *chaves* e *valores*, para um conjunto finito de chaves.

Exemplo: Podemos agrupar numa classe de construtores de tipos as operações que devem estar definidas sobre funções finitas.

```
class FFinite ff where
  val :: (Eq a) => a -> (ff a b) -> Maybe b
  acr :: (Eq a) => (a,b) -> (ff a b) -> (ff a b)
  def :: (Eq a) => a -> (ff a b) -> Bool
  dom :: (Eq a) => (ff a b) -> [a]

  def x t = case (val x t) of
    Nothing -> False
    (Just _) -> True
```

Exemplo: Tabelas implementando listas de associações (chave,valor) podem ser declaradas como instância da classe `FFinite`.

```
data (Eq a) => Tab a b = Tab [(a,b)]
  deriving Show
```

É possível usar o mesmo nome para o construtor de tipo e para o construtor de valores.

148

```
instance FFinite Tab where
  val x (Tab []) = Nothing
  val x (Tab ((c,v):xs)) = if x==c then Just v
                           else val x (Tab xs)

  acr (x,y) (Tab []) = Tab [(x,y)]
  acr (x,y) (Tab ((c,v):t)) = if x==c
                              then Tab ((x,y):t)
                              else let (Tab w) = acr (x,y) (Tab t)
                                   in Tab ((c,v):w)

  dom (Tab t) = map fst t
```

Exercício:

- Defina um tipo de dados polimórfico que implemente listas de associações em árvores binárias e que possa ser instância da classe `FFinite`.
- Declare o construtor do tipo que acabou de definir como instância da classe `FFinite`.

149

Mónades

Na programação funcional, conceito de **mónade** é usado para sintetizar a ideia de **computação**.

Uma **computação** é vista como algo que se passa dentro de uma **“caixa negra”** e da qual conseguimos apenas ver os resultados.

Em Haskell, o conceito de mónade está definido como uma classe de construtores de tipos.

```
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
```

- O termo **(return x)** corresponde a uma computação nula que retorna o valor **x**.
- O operador **(>>=)** corresponde de alguma forma à composição de computações.

150

A classe Monad

```
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
```

- O termo **(return x)** corresponde a uma computação nula que retorna o valor **x**. **return** faz a transição do mundo dos valores para o mundo das computações.
- O operador **(>>=)** corresponde de alguma forma à composição de computações.
- O operador **(>>)** corresponde a uma composição de computações em que o valor devolvido pela primeira computação é ignorado.

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: uma acção de *input/output*, o tratamento de excepções, uma acção sobre o estado, etc.

151

Input / Output

Como conciliar o princípio de “computação por cálculo” com o input/output ?

Que tipos poderão ter as funções de input/output ?

Será que funções para ler um caracter do teclado, ou escrever um caracter no écran, podem ter os seguintes tipos ?

`lerChar :: Char`

É uma constante ?

`escreveChar :: Char -> ()`

Como diferenciar da função `f _ = ()` ?

Em Haskell, existe pré-definido o **construtor de tipos IO**, e é uma instância da classe **Monad**.

Os tipos acima sugeridos estão errados. Essas funções estão pré-definidas e têm os seguintes tipos:

`getChar :: IO Char`

`getChar` é um valor do tipo `Char` que pode resultar de alguma acção de input/output.

`putChar :: Char -> IO ()`

`putChar` é uma função que recebe um caracter e executa alguma acção de input/output, devolvendo `()`.

152

O mónade IO

O mónade IO agrupa os tipos de todas as computações onde existem acções de input/output.

`return :: a -> IO a` é a função que recebe um argumento `x`, não faz qualquer operação de IO, e retorna o mesmo valor `x`.

`(>>=) :: IO a -> (a -> IO b) -> IO b` é o operador que recebe como argumento um programa `p`, que faz algumas operações de IO e retorna um valor `x`, e uma função `f` que “transporta” esse valor para a próxima sequência de operações de IO.

`p >>= f` é o programa que faz as operações de IO correspondentes a `p` seguidas das operações de IO correspondentes a `f x`, retornando o resultado desta última computação.

Exemplo: As seguintes funções já estão pré-definidas.

```
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))
                        )
```

153

A notação “do”

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

`e1 >> e2` pode ser escrito como `do { e1; e2 }` ou `do e1
e2`

`e1 >>= (\x -> e2)` pode ser escrito como `do x <- e1
e2`

`c1 >>= (\x1-> c2 >>= (\x2-> ... cn >>= (\xn-> return y) ...))`

pode ser escrito como `do x1 <- c1
x2 <- c2
...
xn <- cn
return y`

Mais formalmente:

```
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
```

154

A notação “do”

Exemplo: As funções pré-definidas `putStr` e `getLine`, usando a notação “do”.

```
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: Misturando “do” e “let”.

```
test :: IO ()
test = do x <- getLine
        let a = map toUpper x
            b = map toLower x
        putStr a
        putStr "\t"
        putStr b
        putStr "\n"
```

```
> test
aEIou
AEIOU aeiou
>
```

155

Exemplos com IO

Exemplo:

```
expTrig :: IO ()
expTrig = do putStr "Indique um numero: "
            n <- getLine
            let x = ((read n)::Double)
                s = sin x
                c = cos x
            putStr ("0 seno de "+n++" e' "++(show s)++['.', '\n'])
            putStr ("0 coseno de "+n++" e' "++(show c)++".\n")
```

```
> expTrig
Indique um numero: 2.5
0 seno de 2.5 e' 0.5984721.
0 coseno de 2.5 e' -0.8011436.
```

```
> expTrig
Indique um numero: 3.4.5
0 seno de 3.4.5 e' *** Exception: Prelude.read: no parse
```

156

Exemplo:

Uma função que recebe uma listas de questões e vai recolhendo respostas para uma lista.

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

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

Ou, de forma equivalente:

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

157

Funções de IO do Prelude

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

```
getChar  :: IO Char    lê um caracter;
getLine  :: IO String  lê uma string (até se primir enter).
```

Para escrever no *standard output* (por defeito, o écran):

```
putChar  :: Char -> IO ()   escreve um caracter;
putStr   :: String -> IO () escreve uma string;
putStrLn :: String -> IO () escreve uma string e muda de linha;
print    :: Show a => a -> IO () equivalente a (putStrLn . show)
```

Para lidar com ficheiros de texto:

```
writeFile :: FilePath -> String -> IO ()   escreve uma string no ficheiro;
appendFile :: FilePath -> String -> IO ()  acrescenta no final do ficheiro;
readFile  :: FilePath -> IO String         lê o conteúdo do ficheiro para
                                           uma string.
```

```
type FilePath = String
```

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

O **módulo IO** contém outras funções mais sofisticadas de manipulação de ficheiros.

158

```

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

```

```

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 <- return ((read a)::Float)
     putStr "Indique o valor do ceoficiente b: "
     b <- getLine
     b1 <- return ((read b)::Float)
     putStr "Indique o valor do ceoficiente c: "
     c <- getLine
     c1 <- return ((read c)::Float)
     case (roots (a1,b1,c1)) of
       Nothing      -> putStrLn "Nao ha' raizes reais."
       (Just (r1,r2)) -> putStrLn ("As raizes sao "++(show r1)
                                   ++" e "++(show r2))

```

159

O Prelude tem já definida a função `readIO`

```

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))

```

160

Exemplo:

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

```
texto = "1234\tPedro\t15\t17\n1111\tAna\t16\t13\n"
```

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

```
geraNotas :: [[String]] -> Notas
geraNotas ([x,y,z,w]:t) = let x1 = (read x)::Integer
                             z1 = (read z)::Int
                             w1 = (read w)::Int
                             in (x1,y,z1,w1):(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,w):t) = (show x) ++ ('\t':y) ++ ('\t':(show z)) ++
                        ('\t':(show w)) ++ "\n" ++ (geraStr t)
```

161

O mónade Maybe

A declaração do construtor de tipos **Maybe** como instância da classe **Monad** é muito útil para trabalhar com **computações parciais**, pois permite fazer a propagação de erros.

```
instance Monad Maybe where
  return x      = Just x
  (Just x) >>= f = f x
  Nothing >>= _ = Nothing
  fail _       = Nothing
```

Exemplo:

```
exemplo :: Int -> Int -> Int -> Maybe Int
exemplo a b c = do x <- return a
                  y <- return b
                  z <- divide x y
                  w <- soma c z
                  return w
```

Podemos simplificar ?

```
divide :: Int -> Int -> Maybe Int
divide _ 0 = Nothing
divide x y = Just (div x y)
```

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

162

Módulos

Um programa Haskell é uma colecção de **módulos**. A organização de um programa em módulos cumpre dois objectivos:

- criar componentes de software que podem ser usadas em diversos programas;
- dar ao programador algum control sobre os identificadores que podem ser usados.

Um módulo é uma declaração “gigante” que obedece à seguinte sintaxe:

```
module Nome (entidades_a_exportar) where  
  
declarações de importações de módulos  
  
declarações de: tipos, classes, instâncias, assinaturas, funções, ...  
(por qualquer ordem)
```

Cada módulo está armazenado num ficheiro, geralmente com o mesmo nome do módulo, mas isso não é obrigatório.

163

Na declaração de um módulo:

- pode-se indicar explicitamente o conjunto de tipos / construtores / funções / classes que são exportados (i.e., visíveis do exterior)

*Aos vários items que são exportados ou importados chamaremos **entidades**.*

- por defeito, se nada for indicado, todas as declarações feitas do módulo são exportadas;
- é possível exportar um tipo algébrico com os seus construtores fazendo, por exemplo: `ArvBin(Vazia, Nodo)`, ou equivalentemente, `ArvBin(..)`;
- também é possível exportar um tipo algébrico e não exportar os seus construtores, ou exportar apenas alguns;
- os métodos de classe podem ser exportados seguindo o estilo usado na exportação de construtores, ou como funções comuns;
- declarações de instância são sempre exportadas e importadas, por defeito;
- é possível exportar entidades que não estão directamente declaradas no módulo, mas que resultam de alguma importação de outro módulo.

Qualquer entidade visível no módulo é passível de ser exportada por esse módulo.

164

Na importação de um módulo por outro módulo:

- é possível fazer a importação de todas as entidades exportadas pelo módulo fazendo

```
import Nome_do_módulo
```

- é possível indicar explicitamente as entidades que queremos importar, fazendo

```
import Nome_do_módulo (entidades a importar)
```

- é possível indicar selectivamente as entidades que não queremos importar (importa-se tudo o que é exportado pelo outro módulo excepto o indicado)

```
import Nome_do_módulo hiding (entidades a não importar)
```

- é possível fazer com que as entidades importadas sejam referenciadas indicando o módulo de onde provêm como prefixo (seguido de '.') fazendo

```
import qualified Nome_do_módulo (entidades a importar)
```

*(Pode ser útil para evitar **colisões** de nomes, pois é ilegal importar entidades diferentes que tenham o mesmo nome. Mas se for o mesmo objecto que é importado de diferentes módulos, não há colisão. Uma entidade pode ser importada via diferentes caminhos sem que haja conflitos de nomes.)*

165

Um exemplo com módulos

Considere os módulos: [Listas](#), [Arvores](#), [Tempo](#), [Horas](#) e [Main](#), que pretendem ilustrar as diferentes formas de exportar e importar entidades.

```
module Listas where

soma [] = 0
soma (x:xs) = x + (soma xs)

conta = length

naLista x [] = False
naLista x (y:ys) = if x==y then True
                   else naLista x ys

mult = product

cauda (_:xs) = xs
```

166

```

module Arvores(ArvBin(Vazia,Nodo), naArv, soma, mult) where

data ArvBin a = Vazia
              | Nodo a (ArvBin a) (ArvBin a)
  deriving Show

conta Vazia = 0
conta (Nodo _ e d) = 1 + (conta e) + (conta d)

soma Vazia = 0
soma (Nodo x e d) = x + (soma e) + (soma d)

mult Vazia = 1
mult (Nodo x e d) = x * (mult e) * (mult d)

naArv :: (Eq a) => a -> ArvBin a -> Bool
naArv _ Vazia = False
naArv x (Nodo y e d) | x==y      = True
                    | otherwise = (naArv x e) || (naArv x d)

```

167

```

module Tempo(Time, horas, minutos, meioDia, cauda) where

import Listas

data Time = Am Int Int
          | Pm Int Int
          | Total Int Int   deriving Show

hValida (Total h m) = 0<=h && h<24 && 0<=m && m<60
hValida (Am h m)    = 0<=h && h<12 && 0<=m && m<60
hValida (Pm h m)    = 0<=h && h<12 && 0<=m && m<60

horas (Am h m)      = h
horas (Pm h m)      = h + 12
horas (Total h m)   = h

minutos (Am h m)    = m
minutos (Pm h m)    = m
minutos (Total h m) = m

meioDia = (Total 12 00)

ex = cauda "experiencia"

```

168

```

module Horas(Hora(..), Tempo(manha)) where

data Hora = AM Int Int
           | PM Int Int

class Tempo a where
    manha :: a -> Bool
    tarde :: a -> Bool
    tarde t = not (manha t)

instance Tempo Hora where
    manha (AM _ _) = True
    manha (PM _ _) = False

```

169

```

module Main where

import Arvores (ArvBin(..), soma, naArv)
import qualified Listas (soma, mult, conta)
import Tempo
import Horas
import Char hiding (toUpper, isDigit)

arv1 = Nodo 5 (Nodo 3 Vazia (Nodo 4 Vazia Vazia))
      (Nodo 2 (Nodo 1 Vazia Vazia) Vazia)

lis1 = [1,2,3,4]

minTotal :: Time -> Int
minTotal t = (horas t)*60 + (minutos t)

testeC = cauda lis1

toUpper :: Num a => ArvBin a -> ArvBin a
toUpper Vazia = Vazia
toUpper (Nodo x e d) = Nodo (x*x) (toUpper e) (toUpper d)

test = map toLower "tesTAnDo"

```

170

Após carregar o módulo `Main`, analise o comportamento do interpretador.

```
*Main> soma arv1
15
*Main> mult arv1
Variable not in scope: `mult'
*Main> conta arv1
Variable not in scope: `conta'
*Main> Listas.soma lis1
10
*Main> mult lis1
Variable not in scope: `mult'
*Main> Listas.mult lis1
24
```

```
*Main> testeC
[2,3,4]
*Main> hValida meioDia
Variable not in scope: `hValida'
```

```
*Main> isDigit 'e'
Variable not in scope: `isDigit'
*Main> isAlpha 'e'
True
*Main> toUpper arv1
Nodo 25 (Nodo 9 Vazia (Nodo 16 Vazia Vazia))
(Nodo 4 (Nodo 1 Vazia Vazia) Vazia)
*Main> test
"testando"
```

```
*Main> minTotal meioDia
720
*Main> minTotal (Am 9 30)
Data constructor not in scope: `Am'
*Main> manha (AM 9 30)
True
*Main> tarde (PM 17 15)
Variable not in scope: `tarde'
```

171

Compilação de programas Haskell

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 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 *Glasgow Haskell Compiler*, pode ser feita executando na shell do sistema operativo o seguinte comando:

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

Exemplo: Usando o último exemplo para testar a compilação de programas definidos em vários módulos, podemos acrescentar ao módulo `Main` a declaração

```
main = putStrLn "OK"
```

Assumindo que este módulo está guardado no ficheiro `Main.hs` podemos fazer a compilação assim:

```
ghc -o testar --make Main
```

Exemplo: Assumindo que o módulo do próximo slide está no ficheiro `roots.hs`, podemos gerar um executável (chamado raizes) fazendo

```
ghc -o raizes --make roots
```

172

```

module Main where

main :: IO ()
main = do calcRoots
    putStrLn "Deseja continuar (s/n) ? "
    x <- getLine
    case (head x) of
        's' -> main
        'S' -> main
        _   -> putStrLn "\n FIM."

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

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

```

173

Tipos Abstractos de Dados

A quase totalidade dos tipos de dados que vimos até aqui são **tipos concretos de dados**, dado que se referem a uma estrutura de dados concreta fornecida pela linguagem.

Exemplos:

```

data ArvBin a = Vazia
              | Nodo a (ArvBin a) (ArvBin a)

type TB = [(Integer,String)]

```

`(ArvBin a)` e `TB` são dois tipos concretos. Sabemos como são constituídos os valores destes tipos e podemos extrair informação ou construir novos valores, por manipulação directa dos construtores de valores destes tipos.

Em contraste, os **tipos abstractos de dados** não estão ligados a nenhuma representação particular. Em vez disso, eles são definidos implicitamente através de um conjunto de operações utilizadas para os manipular.

Exemplo: O tipo `(IO a)` é um tipo abstracto de dados. Não sabemos de que forma são os valores deste tipo. Apenas conhecemos um conjunto de funções para os manipular.

174

Tipos Abstractos de Dados

As assinaturas das funções do tipo abstracto de dados e as suas especificações constituem o **interface** do tipo abstracto de dados. Nem a estrutura interna do tipo abstracto de dados, nem a implementação destas funções são visíveis para o utilizador.

Dada a especificação de um tipo abstracto de dados, as operações que o definem poderão ter **diferentes implementações**, dependendo da estrutura usada na representação interna de dados e dos algoritmos usados.

A utilização de tipos abstractos de dados traz benefícios em termos de **modularidade** dos programas. Alterações na implementação das operações do tipo abstracto não afecta outras partes do programa desde que as operações mantenham o seu tipo e a sua especificação.

Em Haskell, a construção de tipos abstractos de dados é feita utilizando **módulos**.

O módulo onde se implementa o tipo abstracto de dados deve exportar apenas o nome do tipo e o nome das operações que constituem o seu interface. A representação do tipo fica assim escondida dentro do módulo, não sendo visível do seu exterior.

Deste modo, podemos mais tarde alterar a representação do tipo abstracto sem afectar os programas que utilizam esse tipo abstracto.

175

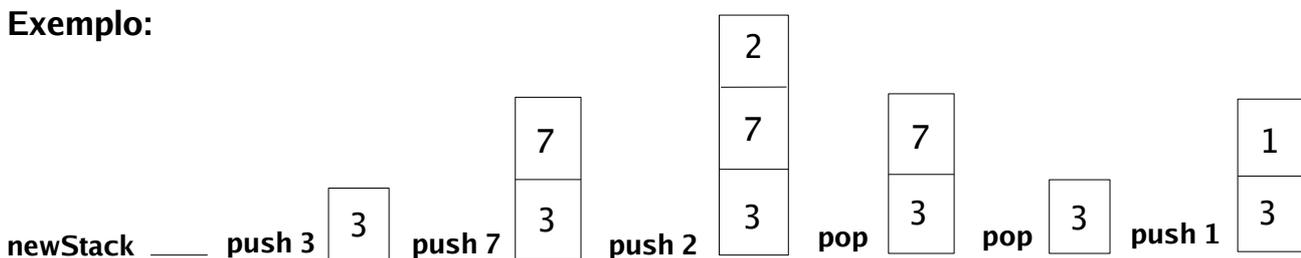
Stacks (pilhas)

Uma **Stack** é uma colecção homogénea de itens que implementa a noção de **pilha**, de acordo com o seguinte interface:

<code>push :: a -> Stack a -> Stack a</code>	coloca um item no topo da pilha
<code>pop :: Stack a -> Stack a</code>	remove o item do topo da pilha
<code>top :: Stack a -> a</code>	dá o item que está no topo da pilha
<code>stackEmpty :: Stack a -> Bool</code>	testa se a pilha está vazia
<code>newStack :: Stack a</code>	cria uma pilha vazia

Os itens da stack são removidos de acordo com a estratégia **LIFO (Last In First Out)**.

Exemplo:



176

```

module Stack(Stack, push, pop, top, stackEmpty, newStack) where

push      :: a -> Stack a -> Stack a
pop       :: Stack a -> Stack a
top       :: Stack a -> a
stackEmpty :: Stack a -> Bool
newStack  :: Stack a

data Stack a = EmptyStk
             | Stk a (Stack a)

push x s = Stk x s

pop EmptyStk = error "pop em stack vazia."
pop (Stk _ s) = s

top EmptyStk = error "top em stack vazia."
top (Stk x _) = x

newStack = EmptyStk

stackEmpty EmptyStk = True
stackEmpty _        = False

instance (Show a) => Show (Stack a) where
  show (EmptyStk) = "#"
  show (Stk x s)  = (show x) ++ "|" ++ (show s)

```

177

```

module Main where

import Stack

listT0stack :: [a] -> Stack a
listT0stack []      = newStack
listT0stack (x:xs) = push x (listT0stack xs)

stackT0list :: Stack a -> [a]
stackT0list s
  | stackEmpty s = []
  | otherwise    = (top s):(stackT0list (pop s))

ex1 = push 2 (push 7 (push 3 newStack))
ex2 = push "abc" (push "xyz" newStack)

```

Exemplos:

```

*Main> ex1
2|7|3|#
*Main> ex2
"abc"|"xyz"|#

```

```

*Main> listT0stack [1,2,3,4,5]
1|2|3|4|5|#
*Main> stackT0list ex2
["abc","xyz"]
*Main> stackT0list (listT0stack [1,2,3,4,5])
[1,2,3,4,5]

```

178

```

module Stack(Stack, push, pop, top, stackEmpty, newStack) where

push      :: a -> Stack a -> Stack a
pop       :: Stack a -> Stack a
top       :: Stack a -> a
stackEmpty :: Stack a -> Bool
newStack  :: Stack a

data Stack a = Stk [a]

push x (Stk s) = Stk (x:s)

pop (Stk []) = error "pop em stack vazia."
pop (Stk (_:xs)) = Stk xs

top (Stk []) = error "top em stack vazia."
top (Stk (x:_)) = x

newStack = Stk []

stackEmpty (Stk []) = True
stackEmpty _ = False

instance (Show a) => Show (Stack a) where
  show (Stk []) = "#"
  show (Stk (x:xs)) = (show x) ++ "|" ++ (show (Stk xs))

```

179

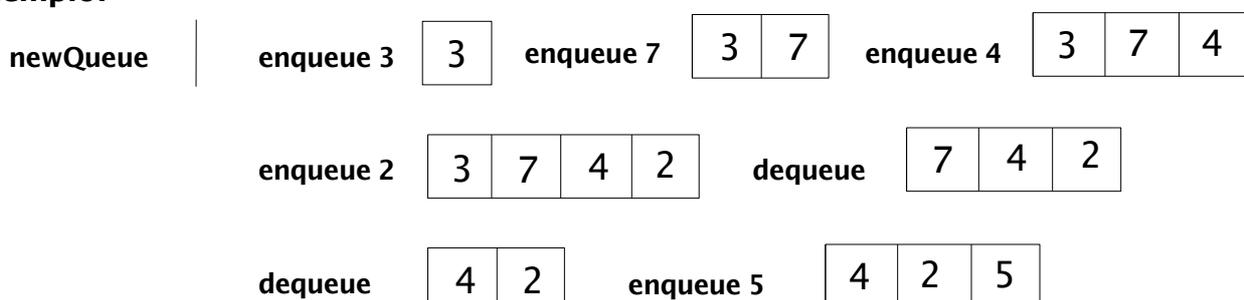
Queues (filas)

Uma **Queue** é uma colecção homogénea de itens que implementa a noção de **fila de espera**, de acordo com o seguinte interface:

<code>enqueue :: a -> Queue a -> Queue a</code>	coloca um item no fim da fila de espera
<code>dequeue :: Queue a -> Queue a</code>	remove o item do início da fila de espera
<code>front :: Queue a -> a</code>	dá o item que está à frente na fila de espera
<code>queueEmpty :: Queue a -> Bool</code>	testa se a fila de espera está vazia
<code>newQueue :: Queue a</code>	cria uma fila de espera vazia

Os itens da queue são removidos de acordo com a estratégia **FIFO (First In First Out)**.

Exemplo:



180

```

module Queue(Queue, enqueue, dequeue, front, queueEmpty, newQueue) where

enqueue    :: a -> Queue a -> Queue a
dequeue    :: Queue a -> Queue a
front      :: Queue a -> a
queueEmpty :: Queue a -> Bool
newQueue   :: Queue a

data Queue a = Q [a]

enqueue x (Q q) = Q (q++[x])

dequeue (Q (_:xs)) = Q xs
dequeue _          = error "Fila de espera vazia."

front (Q (x:_)) = x
front _         = error "Fila de espera vazia."

queueEmpty (Q []) = True
queueEmpty _     = False

newQueue = (Q [])

instance (Show a) => Show (Queue a) where
  show (Q []) = "."
  show (Q (x:xs)) = "<"++(show x)++(show (Q xs))

```

181

```

module Main where

import Stack
import Queue

queueTostack :: Queue a -> Stack a
queueTostack q = qts q newStack
  where qts q s
    | queueEmpty q = s
    | otherwise    = qts (dequeue q) (push (front q) s)

stackTqueue :: Stack a -> Queue a
stackTqueue s = stq s newQueue
  where stq s q
    | stackEmpty s = q
    | otherwise    = stq (pop s) (enqueue (top s) q)

invQueue :: Queue a -> Queue a
invQueue q = stackTqueue (queueTostack q)

invStack :: Stack a -> Stack a
invStack s = queueTostack (stackTqueue s)

q1 = enqueue 3 (enqueue 6 (enqueue 1 newQueue))
s1 = push 2 (push 8 (push 9 newStack))

```

182

Exemplos:

```
*Main> q1
<1<6<3.
*Main> queueTostack q1
3|6|1|#
*Main> invQueue q1
<3<6<1.
```

```
*Main> s1
2|8|9|#
*Main> stackToqueue s1
<2<8<9.
*Main> invStack s1
9|8|2|#
```

183

Sets (conjuntos)

Um **Set** é uma colecção homogénea de itens que implementa a noção de **conjunto**, de acordo com o seguinte interface:

<code>emptySet :: Set a</code>	cria um conjunto vazio
<code>setEmpty :: Set a -> Bool</code>	testa se um conjunto é vazio
<code>inSet :: (Eq a) => a -> Set a -> Bool</code>	testa se um item pertence a um conjunto
<code>addSet :: (Eq a) => a -> Set a -> Set a</code>	acrescenta um item a um conjunto
<code>delSet :: (Eq a) => a -> Set a -> Set a</code>	remove um item de um conjunto
<code>pickSet :: Set a -> a</code>	escolhe um item de um conjunto

É necessário testar a igualdade entre itens, por isso o tipo dos itens tem que pertencer à classe `Eq`. Mas certas implementações do tipo `Set` podem requerer outras restrições de classe sobre o tipo dos itens.

É possível estabelecer um interface mais rico para o tipo abstracto `Set`, por exemplo, incluindo operações de **união**, **intersecção** ou **diferença** de conjuntos, embora se consiga definir estas operações à custa do interface actual.

A seguir apresentam-se duas implementações para o tipo abstracto `Set`.

184

```

module Set(Set, emptySet, setEmpty, inSet, addSet, delSet) where

emptySet :: Set a
setEmpty :: Set a -> Bool
inSet    :: (Eq a) => a -> Set a -> Bool
addSet   :: (Eq a) => a -> Set a -> Set a
delSet   :: (Eq a) => a -> Set a -> Set a
pickSet  :: Set a -> a

data Set a = S [a]    -- listas com repetições

emptySet = S []

setEmpty (S []) = True
setEmpty _      = False

inSet _ (S [])           = False
inSet x (S (y:ys)) | x == y = True
                  | otherwise = inSet x (S ys)

addSet x (S s) = S (x:s)

delSet x (S s) = S (delete x s)

delete x [] = []
delete x (y:ys) | x == y = delete x ys
                | otherwise = y:(delete x ys)

pickSet (S []) = error "Conjunto vazio"
pickSet (S (x:_)) = x

```

185

```

module Set(Set, emptySet, setEmpty, inSet, addSet, delSet) where

emptySet :: Set a
setEmpty :: Set a -> Bool
inSet    :: (Eq a) => a -> Set a -> Bool
addSet   :: (Eq a) => a -> Set a -> Set a
delSet   :: (Eq a) => a -> Set a -> Set a
pickSet  :: Set a -> a

data Set a = S [a]    -- listas sem repetições

emptySet = S []

setEmpty (S []) = True
setEmpty _      = False

inSet _ (S [])           = False
inSet x (S (y:ys)) | x == y = True
                  | otherwise = inSet x (S ys)

addSet x (S s) | (elem x s) = S s
                | otherwise = S (x:s)

delSet x (S s) = S (delete x s)

delete x [] = []
delete x (y:ys) | x == y = ys
                | otherwise = y:(delete x ys)

pickSet (S []) = error "Conjunto vazio"
pickSet (S (x:_)) = x

```

186

Tables (tabelas)

(Table a b) é uma colecção de associações entre **chaves** do tipo **a** e **valores** do tipo **b**, implementando assim uma função finita, com domínio em **a** e co-domínio em **b**, através de uma determinada estrutura de dados.

O tipo abstracto **tabela** poderá ter o seguinte interface:

```
newTable :: Table a b
```

```
findTable :: (Ord a) => a -> Table a b -> Maybe b
```

```
updateTable :: (Ord a) => (a,b) -> Table a b -> Table a b
```

```
removeTable :: (Ord a) => a -> Table a b -> Table a b
```

Para permitir implementações eficientes destas operações, está-se a exigir que o tipo das chaves pertença à classe `Ord`.

A seguir apresentam-se duas implementações distintas para o tipo abstracto `tabela`:

- usando uma lista de pares (*chave,valor*) ordenada por ordem crescente das chaves;
- usando uma árvore binária de procura com pares (*chave, valor*) nos nodos da árvore.

187

```
module Table(Table, newTable, findTable, updateTable, removeTable) where

newTable    :: Table a b
findTable   :: (Ord a) => a -> Table a b -> Maybe b
updateTable :: (Ord a) => (a,b) -> Table a b -> Table a b
removeTable :: (Ord a) => a -> Table a b -> Table a b

data Table a b = Tab [(a,b)]    -- lista ordenada por ordem crescente

newTable = Tab []

findTable _ (Tab []) = Nothing
findTable x (Tab ((c,v):cvs))
  | x < c = Nothing
  | x == c = Just v
  | x > c = findTable x (Tab cvs)

updateTable (x,z) (Tab []) = Tab [(x,z)]
updateTable (x,z) (Tab ((c,v):cvs))
  | x < c = Tab ((x,z):(c,v):cvs)
  | x == c = Tab ((c,z):cvs)
  | x > c = let (Tab t) = updateTable (x,z) (Tab cvs)
            in Tab ((c,v):t)
```

{- -- continua -- -}

188

{- -- continuação do slide anterior -- -}

```
removeTable _ (Tab []) = Tab []
removeTable x (Tab ((c,v):cvs))
    | x < c = Tab ((c,v):cvs)
    | x == c = Tab cvs
    | x > c = let (Tab t) = removeTable x (Tab cvs)
              in Tab ((c,v):t)

instance (Show a, Show b) => Show (Table a b) where
    show (Tab []) = ""
    show (Tab ((c,v):cvs)) = (show c)++"\t"++(show v)++"\n"++(show (Tab cvs))
```

Evita-se derivar o método show de forma automática, para não revelar a implementação do tipo abstracto.

189

```
module Table(Table, newTable, findTable, updateTable, removeTable) where

newTable    :: Table a b
findTable   :: (Ord a) => a -> Table a b -> Maybe b
updateTable :: (Ord a) => (a,b) -> Table a b -> Table a b
removeTable :: (Ord a) => a -> Table a b -> Table a b

-- Arvore binaria de procura
data Table a b = Empty
               | Node (a,b) (Table a b) (Table a b)

newTable = Empty

findTable _ Empty = Nothing
findTable x (Node (c,v) e d)
    | x < c = findTable x e
    | x == c = Just v
    | x > c = findTable x d

updateTable (x,z) Empty = Node (x,z) Empty Empty
updateTable (x,z) (Node (c,v) e d)
    | x < c = Node (c,v) (updateTable (x,z) e) d
    | x == c = Node (c,z) e d
    | x > c = Node (c,v) e (updateTable (x,z) d)
```

{- -- continua -- -}

190

{- -- continuação do slide anterior -- -}

```
removeTable _ Empty = Empty
removeTable x (Node (c,_) e Empty) | x == c = e
removeTable x (Node (c,_) Empty d) | x == c = d
removeTable x (Node (c,v) e d)
    | x < c = Node (c,v) (removeTable x e) d
    | x > c = Node (c,v) e (removeTable x d)
    | x == c = let (y,z) = minTable d
                in Node (y,z) e (removeTable y d)

minTable :: Table a b -> (a,b)
minTable (Node (c,v) Empty _) = (c,v)
minTable (Node _ e _) = minTable e

instance (Show a, Show b) => Show (Table a b) where
    show Empty = ""
    show (Node (c,v) e d) = (show e)++(show c)++"\t"++(show v)++"\n"++(show d)
```

191

```
module Main where

import Table

type Numero = Integer
type Nome    = String
type Nota    = Integer

pauta :: [(Numero, Nome, Nota)] -> Table Numero (Nome, Nota)
pauta [] = newTable
pauta ((x,y,z):xyzs) = updateTable (x,(y,z)) (pauta xys)

info = [(1111, "Mario", 14), (5555, "Helena", 15), (3333, "Teresa", 12),
        (7777, "Pedro", 15), (2222, "Rui", 17), (9999, "Pedro", 10)]
```

Exemplos:

```
*Main> pauta info
1111 ("Mario",14)
2222 ("Rui",17)
3333 ("Teresa",12)
5555 ("Helena",15)
7777 ("Pedro",15)
9999 ("Pedro",10)
```

```
*Main> findTable 5555 (pauta info)
Just ("Helena",15)
*Main> findTable 8888 (pauta info)
Nothing
*Main> removeTable 9999 (pauta info)
1111 ("Mario",14)
2222 ("Rui",17)
3333 ("Teresa",12)
5555 ("Helena",15)
7777 ("Pedro",15)
```

Como estará a tabela implementada ?

192

Sequência de Fibonacci

O n-ésimo número da sequência de Fibonacci define-se matematicamente por

$$\begin{aligned} fib\ n &= 0 && ,\ se\ n = 0 \\ fib\ n &= 1 && ,\ se\ n = 1 \\ fib\ n &= fib\ (n-2) + fib\ (n-1) && ,\ se\ n \geq 2 \end{aligned}$$

```
fib 0 = 0
fib 1 = 1
fib n | n >= 2 = fib (n-2) + fib (n-1)
```

O cálculo do fib de um número pode envolver o cálculo do fib de números mais pequenos, repetidas vezes.

$$\begin{aligned} fib\ 5 &\Rightarrow (fib\ 3) + (fib\ 4) \Rightarrow ((fib\ 1) + (fib\ 2)) + ((fib\ 2) + (fib\ 3)) \\ &\Rightarrow (1 + ((fib\ 0) + (fib\ 1))) + ((fib\ 2) + (fib\ 3)) \Rightarrow \dots \Rightarrow \dots \Rightarrow 5 \end{aligned}$$

A sequência de Fibonacci pode ser definida por

```
seqFibonacci = [ fib n | n <- [0,1..] ]
```

193

Uma versão mais eficiente dos números de Fibonacci utiliza um parametro de acumulação.

Neste caso o acumulador é um par que regista os dois últimos números de Fibonacci calculados até ao momento.

```
fib n = fibAc (0,1) n
  where fibAc (a,b) 0 = a
        fibAc (a,b) 1 = b
        fibAc (a,b) (n+1) = fib (b,a+b) n
```

$$\begin{aligned} fib\ 5 &\Rightarrow fibAc\ (0,1)\ 5 \Rightarrow fibAc\ (1,1)\ 4 \Rightarrow fibAc\ (1,2)\ 3 \\ &\Rightarrow fibAc\ (2,3)\ 2 \Rightarrow fibAc\ (3,5)\ 1 \Rightarrow 5 \end{aligned}$$

A sequência de Fibonacci pode ser definida por

```
seqFib = 0 : 1 : [ a+b | (a,b) <- zip seqFib (tail seqFib) ]
```

Note que é a [lazy evaluation](#) que faz com que este género de definição seja possível.

194

Funções e listas por compreensão

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

Exemplo: Máximo divisor comum de dois números.

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

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

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

O crivo de Eratosthenes

Esta função deixa ficar numa lista o primeiro elemento e todos os que **não são** múltiplos desse argumento, repetindo em seguida esta operação para a restante lista.

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

A lista dos números primos não superiores a um dado número.

```
primos_ate' x = crivo [2..x]
```

Lista dos números primos.

```
seqPrimos = crivo [2..]
```

Calcular os n primeiros primos.

```
primeirosPrimos n = take n seqPrimos
```