

Fundamentos da Programação

Primeira aula:
Funções em Haskell

Nesta aula vamos...

- Tomar nota dos objectivos de FP.
- Relembrar algumas banalidades sobre computadores e programação.
- Conhecer a linguagem Haskell, que usaremos neste curso.
- Ilustrar a programação em Haskell resolvendo um pequeno problema de programação.

Nesta aula *não* vamos...

- Perder tempo com a “apresentação” da cadeira
- Para isso, há a página, no Moodle, <http://www.ualg.pt/moodle2007/course/view.php?id=60010024>



Não deixe de ver
a bibliografia.

Objectivos de FP

- Analisar e explicar o comportamento de programas simples
- Modificar e fazer evoluir programas simples.
- Desenhar, implementar, testar e depurar programas simples
- Escolher o algoritmo apropriado para cada tarefa de programação.
- Aplicar a decomposição funcional ao desenvolvimento de software.
- Compreender elementarmente o processo de desenvolvimento de software.

Futuras cadeiras de programação

- Programação Imperativa
- Algoritmos e Estruturas de Dados I e II
- Programação Orientada por Objectos
- Bases de Dados
- Inteligência Artificial
- Computação Gráfica
- Compiladores
- Desenvolvimento de Aplicações para a Web

Programação

- *Programar* é escrever programas para computador.
- Computador = Máquina programável, formada por uma ou várias unidades de processamento, controlada por programas registados internamente, capaz de executar cálculos complexos, que incluem muitas operações aritméticas e muitas operações lógicas, sem intervenção humana.

Programas

- Os programas controlam, ou conduzem, os cálculos que os computadores vão executando.
- Os programas são descrições textuais, feitas usando uma *linguagem de programação*.

Linguagens de Programação

- Cada linguagem de programação é um conjunto de regras sintáticas e semânticas, com um documento de referência.
- Exemplos de linguagens de programação: FORTRAN, Algol, Lisp, COBOL, Basic, Pascal, Prolog, C, Ada, C++, Eiffel, Java, C#, Python, Ruby, Haskell.

Problema: calcular a nota em FP

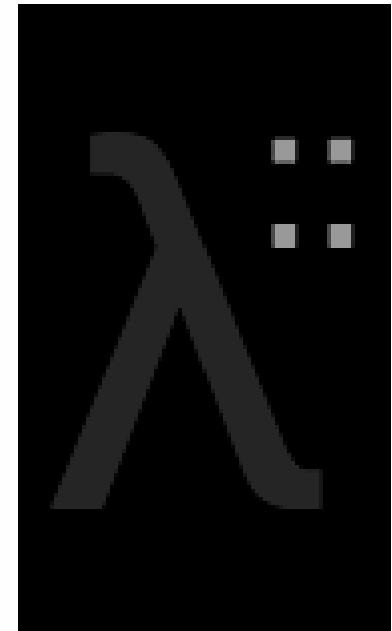
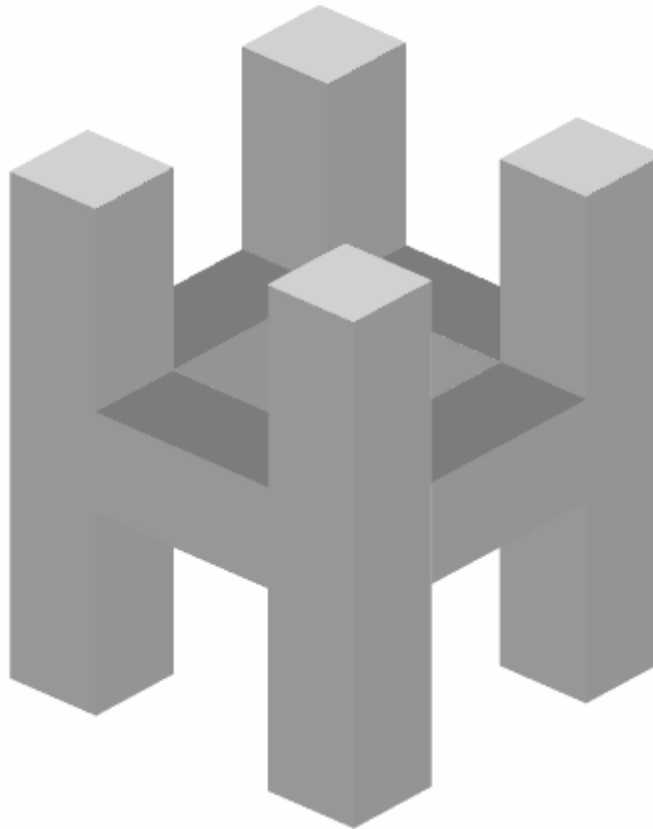
- Regra (simplificada): a nota é a média ponderada da nota do exame com peso 60% e do avaliação distribuída com peso 40%, arredondada às unidades, se a nota do exame for maior ou igual a 8.5, ou é a nota do exame arredondada às unidades, se não.

Exemplos

- Se a nota do exame for 13.5 e a da avaliação distribuída for 16.8 então calculamos $13.5 * 0.6 + 16.8 * 0.4$, o que dá 14.82. Isto arredondado às unidades dá 15.
- Se a nota do exame for 7.3 e a do avaliação distribuída for 14, nem é preciso fazer contas, pois aplica-se a segunda regra e nota é 7.
- Programar isto é escrever um programa para descrever a realização destes cálculos, independentemente dos valores dos dados.

Que linguagem vamos usar?

HASKELL⁹⁸



Simplificando

- Começamos por programar a média ponderada.
- É uma função com dois argumentos que são números reais e cujo resultado é um número real.
- Programamos uma *função* Haskell:

```
-- First argument is exam, second argument is lab.
```

```
weightedAverage :: Double -> Double -> Double
```

```
weightedAverage x y = x * 0.6 + y * 0.4
```

Explicação

Isto é um
comentário.

```
-- First argument is exam, second argument is lab.
```

```
weightedAverage :: Double -> Double -> Double
```

```
weightedAverage x y = x * 0.6 + y * 0.4
```

Isto é a
assinatura
da função.

Isto é a
definição
da função.

Assinatura

```
weightedAverage :: Double -> Double -> Double
```

- A assinatura declara os tipos dos argumentos e do resultado da função.
- Neste caso, há dois argumentos, ambos números reais, e o resultado também é um número real.
- Os números reais são representados pelo tipo Double.

Definição

`weightedAverage x y = x * 0.6 + y * 0.4`

- A definição exprime os cálculos, usando regras muito semelhantes às da matemática.
- Diferença notável: os argumentos das funções não vêm entre parêntesis, nem na definição, nem na chamada. Escrever `weightedAverage (x, y)` estaria mal.

Experimentando

```
WinHugs
File Edit Actions Browse Help
[Icons]

Hugs 98: Bas
Copyright (c
World wide w
Bugs: http://

Version: Sep 2006

Haskell 98 mode: Restart with command 1

Type :? for help
Hugs> :load gradesFP.hs
Main> weightedAverage 13.5 16.8
14.82
Main> weightedAverage 12 12
12.0
Main> weightedAverage 40 60
48.0
Main> weightedAverage 1000 2000
1400.0
Main>
```

gradesFP.hs
é o ficheiro
que contém a
função

Experimentando, com erros

```
Wnhugs
File Edit Actions Browse Help
[Icons]
Hugs 98: Based on the Haskell 98 standard
Copyright (c) 1994-2005
world wide web: http://haskell.org/hugs
Bugs: http://hackage.haskell.org/trac/hugs
Version: Sep 2006

Haskell 98 mode: Restart with command line option -98 to enable extensions

Type :? for help
Hugs> :load gradesFP.hs
Main> weightedAverage 13.5 16.8
14.82
Main> weightedAverage 12 15
13.2
Main> weightedAverage (13.5, 16.8)
ERROR - type error in application
*** Expression   : weightedAverage (13.5,16.8)
*** Term        : (13.5,16.8)
*** Type        : (a,b)
*** Does not match : Double

Main> weightedAverage 16.3
ERROR - Cannot find "show" function for:
*** Expression : weightedAverage 16.3
*** Of type    : Double -> Double

Main> weightedAverage 12.1 13.5 19.2
ERROR - type error in application
*** Expression   : weightedAverage 12.1 13.5 19.2
*** Term        : weightedAverage
*** Type        : Double -> Double -> Double
*** Does not match : a -> b -> c -> d

Main>
```

Calculadora funcional

```
Haskell 98 mode: Restart with command line option -98
to enable extensions

Type :? for help
Hugs> weightedAverage 12 14
ERROR - Undefined variable "weightedAverage"
Hugs> 3+4
7
Hugs> :load gradesFP.hs
Main> weightedAverage 12 14
12.8
Main> 2 * p1
ERROR - Undefined variable "p1"
Main> 2 * pi
6.28318530717959
Main> sum[4, 6, 8, 1]
19
Main> product[1,2,3,4,5,6]
720
Main> product[1..6]
720
Main> product[1..100]
933262154439441526816992388562667004907159682643816214
685929638952175999932299156089414639761565182862536979
208272237582511852109168640000000000000000000000000
Main>
```

Na verdade, o Hugs é uma calculadora funcional.

Calcula com as nossas funções, que carregou do ficheiro, e com as que já lá tem no Prelúdio.

Controlando a nota mínima no exame

- Deixemos os arredondamentos para o fim.
- Se a nota do exame for menor do que 8.5, é essa a nota final:

```
gradeExact :: Double -> Double -> Double
gradeExact x y
    | x < 8.5 = x
    | otherwise = weightedAverage x y
```

Isto vem no ficheiro
gradesFP.hs

Recorremos à
outra função

Testando

- Usamos a calculadora Hugs para testar com vários casos exemplares:

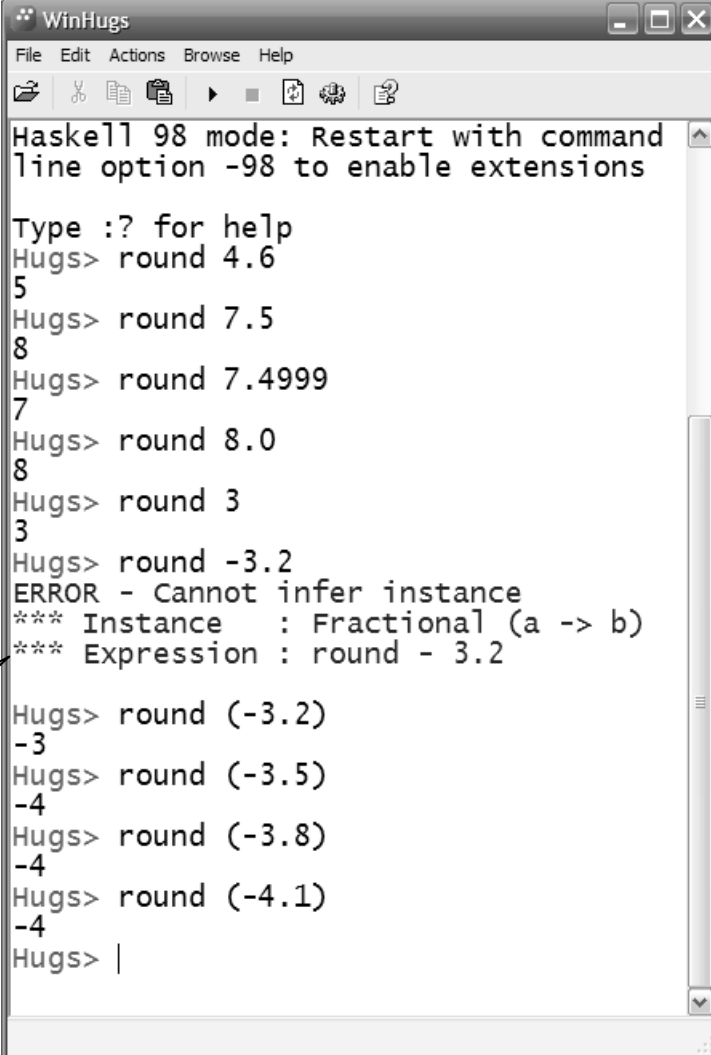
```
Haskell 98 mode: Restart with command  
to enable extensions  
  
Type :? for help  
Hugs> :load "gradesFP.hs"  
Main> gradeExact 13.5 16.8  
14.82  
Main> gradeExact 7.7 18  
7.7  
Main> gradeExact 8.5 18  
12.3  
Main> gradeExact 8.49 10  
8.49  
Main> gradeExact 0 20  
0.0  
Main> gradeExact 20 0  
12.0  
Main> |
```

Arredondamentos

Para arredondar, usamos a função `round`, que está no Prelúdio.

Primeiro, experimentemos a função `round`:

Aqui, sem os parêntesis, o Hugs tentaria fazer uma subtracção



```
WinHugs
File Edit Actions Browse Help
Haskell 98 mode: Restart with command
line option -98 to enable extensions
Type :? for help
Hugs> round 4.6
5
Hugs> round 7.5
8
Hugs> round 7.4999
7
Hugs> round 8.0
8
Hugs> round 3
3
Hugs> round -3.2
ERROR - Cannot infer instance
*** Instance   : Fractional (a -> b)
*** Expression : round - 3.2
Hugs> round (-3.2)
-3
Hugs> round (-3.5)
-4
Hugs> round (-3.8)
-4
Hugs> round (-4.1)
-4
Hugs> |
```

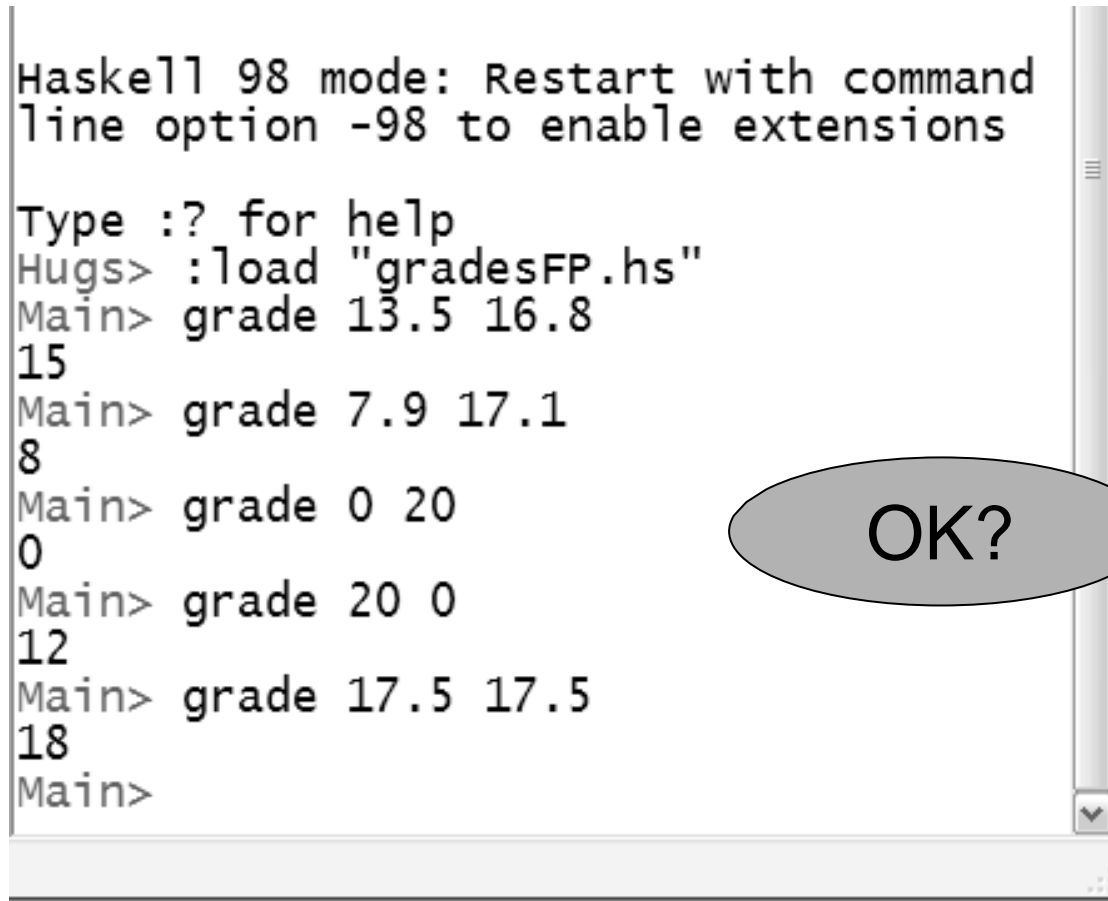
A nota final

- Arredondamos a nota exacta, e pronto:

```
grade :: Double -> Double -> Int  
grade x y = round (gradeExact x y)
```

- O tipo dos números inteiros é Int.
- Os parêntesis são precisos porque senão o argumento do round seria gradeExact, o que não faz sentido.

Testando, finalmente



```
Haskell 98 mode: Restart with command
line option -98 to enable extensions

Type :? for help
Hugs> :load "gradesFP.hs"
Main> grade 13.5 16.8
15
Main> grade 7.9 17.1
8
Main> grade 0 20
0
Main> grade 20 0
12
Main> grade 17.5 17.5
18
Main>
```

OK?

Exercícios

- Programe a função que converte de graus Celsius para graus Fahrenheit e vice-versa?
- Modifique a função da nota de maneira a que todos os 9 na nota final sejam transformados em 10.
- E se quem tiver menos do que 7.5 na avaliação contínua não puder passar, ficando com a nota calculada como anteriormente se der menos do que 9, ou 9, se não?

Controlo

1. Que sinal se usa a seguir ao nome da função na assinatura?
2. De acordo com os exemplos, os nomes das funções escrevem-se com maiúscula inicial ou com minúscula inicial? E os nomes dos tipos?
3. Que tipos vimos nesta aula?
4. Em que circunstâncias foi preciso usar parêntesis na definição de funções?
5. Como se chama a função que arredonda?

Na próxima aula...

- Estudaremos com mais pormenor, os tipos de números de que o Haskell dispõe e as operações com eles.

Fundamentos da Programação

Segunda aula:
Tipos básicos: Inteiros e Booleanos

Nesta aula vamos...

- Estudar alguns dos tipos básicos em Haskell.
- Os tipos básicos são os mesmos em quase todas as linguagens: representam números (inteiros e reais), valores lógicos, caracteres individuais e cadeias de caracteres.
- Mais tarde, com estes tipos construiremos outros, compostos.
- Hoje vamos ver os tipos para números inteiros e para valores lógicos.

O tipo Int

- Os números inteiros são representados pelo tipo Int.
- Mas atenção, só os números entre -2^{31} e $2^{31}-1$ figuram no tipo Int.
- Para números fora desse intervalo, usaremos o tipo Integer.
- Quanto vale $2^{31}-1$?

2147483647

```
Haskell 98 mode: Restart with  
command line option -98 to enable  
extensions
```

```
Type :? for help  
Hugs> 2^31 - 1  
2147483647  
Hugs>
```

Um pouco mais
do que
dois mil milhões.

- O tipo `Int` tem os operadores aritméticos habituais: `+`, `-`, `*`, ``div``, ``mod`` e `^`.
- Também as funções `even`, `odd`, `abs`.

Exercício: programar a multiplicação de inteiros

- Palavras para quê?

```
prod :: Int -> Int -> Int
prod x y
  | y == 0 = 0
  | otherwise = x + prod x (y-1)
```

- Experiência:

```
Main> prod 5 6
30
Main> prod 99 100
9900
Main> prod 55 0
0
Main> prod 0 40
0
Main> prod 25 25
625
Main> |
```

OK?

Where?

- Só para ilustrar a utilização da cláusula where, observe esta outra maneira de programar:

```
prod' :: Int -> Int -> Int
prod' x y
  | y == 0 = 0
  | otherwise = x + prod' x y'
  where y' = y - 1
```

O where é útil quando a subexpressão é complicada ou aparece mais do que uma vez (o que não é o caso aqui ☹)

Ah, e os números negativos?

```
Main> prod 4 -7
ERROR - Cannot infer instance
*** Instance      : Num (Int -> Int)
*** Expression   : prod 4 - 7

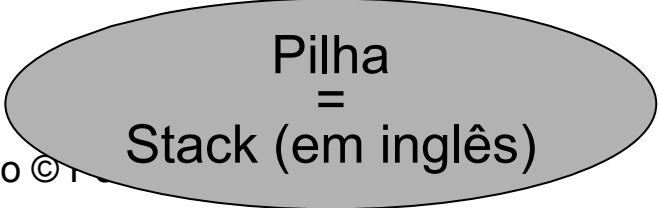
Main> |
```

- Hmm, o Haskell “pensa” que estamos a tentar multiplicar 4 pelo sinal menos, como se tivéssemos escrito (prod 4 -) 7.
- Logo, devemos é escrever prod 4 (-7).

Stack Overflow

```
Main> prod 4 (-7)
ERROR - C stack overflow
Main> |
```

- Analisando a execução verificamos que `prod 4 (-7)` é `4 + prod 4 (-8)` ou seja, `4 + 4 + prod 4 (-9)` ou seja `4 + 4 + 4 + prod 4 (-10)`, etc. Isto nunca mais pára.
- Ou, melhor, pára quando se esgotar pilha de execução.
- A pilha de execução é a zona de memória que o computador usa para os cálculos do nosso programa.



Pilha
=
Stack (em inglês)

Stack overflow, bis

- Na verdade, às vezes a pilha esgota-se antes de os cálculos terminarem, mesmo quando os cálculos não seriam infinitos:

```
Main> prod 1 10
```

```
10
```

```
Main> prod 1 100
```

```
100
```

```
Main> prod 1 1000
```

```
1000
```

```
Main> prod 1 10000
```

```
10000
```

```
Main> prod 1 100000
```

```
ERROR - C stack overflow
```

```
Main>
```

Mas, multiplicando ao contrário, tudo bem!

```
Main> prod 100000 1  
100000
```

```
Main>
```

Outra maneira de multiplicar

- Sabemos que $x*y$ é o mesmo que $(2*x)*(y/2)$ se y for par ou isso mais x , se y for ímpar.
- Em Haskell, o quociente de x por y , para x e y de tipo `Int`, escreve-se `x `div` y` ou `div x y`.

```
prodFast :: Int -> Int -> Int
prodFast x y
  | y == 0 = 0
  | otherwise = prodFast (2 * x) (y `div` 2) + if odd y then x else 0
```

Claro que isto é batota, pois estamos definir a multiplicação e a usar `*` e `div`. No entanto, multiplicar por 2 e dividir por 2 são operações básicas nos computadores.

Com where e fromEnum

- Só para ilustrar outras técnicas:

```
prodFast2 :: Int -> Int -> Int
prodFast2 x y
  | y == 0 = 0
  | otherwise = prodFast2 x' y' + z
  where
    x' = 2 * x
    y' = div y 2
    z = x * fromEnum (odd y)
```

```
Main> fromEnum False
0
Main> fromEnum True
1
Main>
```

Expressão condicional

- O valor da expressão condicional

```
if expr1 then expr2 else expr3
```

expr1 é de tipo Bool e expr2
e expr3 são do mesmo tipo.

é o valor de expr2 se expr1 valer True e
o de expr3 se expr1 valer False.

- Logo, if odd y then x else 0 vale x se y
for ímpar e vale 0 se não.

O tipo Bool

- O tipo Bool contém os valores lógicos, True e False.
- As operações lógicas são &&, || e not.
- Exercício: programar a função odd:

```
odd' :: Int -> Bool  
odd' x = mod x 2 == 1
```

Por vezes, é mais prático escrever `mod x y` do que `x `mod` y`. Tem o mesmo significado. Analogamente para `div x y` e `x `div` y`.

Exemplo: ano bissexto

- Um ano é bissexto se for múltiplo de 4 e não for múltiplo de 100, excepto se for múltiplo de 400:

```
isLeapYear :: Int -> Bool
```

```
isLeapYear x = mod x 400 == 0 ||
```

```
mod x 4 == 0 && mod x 100 /= 0
```

A prioridade de && é mais alta do que a de ||.

Esta é clássica. Não se esqueça dela.

```
Main> isLeapYear 2007
False
Main> isLeapYear 2008
True
Main> isLeapYear 2000
True
Main> isLeapYear 1900
False
Main> isLeapYear 2100
False
Main> |
```


Testando o produto rápido

```
Main> prodFast 1 10000
10000
Main> prodFast 1 100000
100000
Main> prodFast 1 1000000
1000000
Main> prodFast 1 10000000
10000000
Main> prodFast 1 100000000
100000000
Main> prodFast 1 1000000000
1000000000
Main> prodFast 1 10000000000
10000000000
Main> prodFast 1 100000000000
Program error: arithmetic overflow
Main>
```

- Oops: usámos um número maior que 2147483647.

Inteiros de precisão arbitrária

- O tipo Integer representa os números inteiros, com um precisão arbitrária (mas não infinita, claro...)
- De resto, usa-se como tipo Int.
- Exemplo: produto rápido para números Integer:

```
prodFast' :: Integer -> Integer -> Integer
prodFast' x y
  | y == 0 = 0
  | otherwise = prodFast' (2 * x) (y `div` 2) + if odd y then x else 0
```

Calculando com Integer

```
Main> prodFast' 1 100000000
100000000
Main> prodFast' 1 1000000000
1000000000
Main> prodFast' 1 10000000000
10000000000
Main> prodFast' 1 1000000000000000000000
1000000000000000000000
Main> prodFast' 947597994636454 66112621622
62648187669165868749808388
Main>
```

- Repare:

```
Main> 77::Int
77
Main> 77::Integer
77
Main> 457989299345923::Int

Program error: arithmetic overflow

Main> 457989299345923::Integer
457989299345923
Main>
```

Este número não
existe no tipo Int.

O factorial

- O factorial de um número inteiro não negativo x é o produto de todos os números inteiros entre 1 e x .
- Programa-se assim, por exemplo:

```
fact :: Int -> Int
```

OK

```
fact x
```

```
  | x == 0 = 1
```

KO

```
  | otherwise = x * fact (x-1)
```

```
Main> fact 5
120
Main> fact 0
1
Main> fact 10
3628800
Main> fact 20
-2102132736
Main> fact 30
1409286144
Main> fact 40
0
Main>
```

O factorial grande

- Basta usar Integer

```

Main> factBig 40
815915283247897734345611269596115894272000000000
Main> factBig 30
265252859812191058636308480000000
Main> factBig 20
2432902008176640000
Main> factBig 10
3628800
Main> factBig 5
120
Main> factBig 0
1
Main> factBig 100
93326215443944152681699238856266700490715968264381621468
59296389521759999322991560894146397615651828625369792082
7223758251185210916864000000000000000000000000000000000
Main> factBig 200
78865786736479050355236321393218506229513597768717326329
47425332443594499634033429203042840119846239041772121389
19638830257642790242637105061926624952829931113462857270
76331723739698894392244562145166424025403329186413122742
82948532775242424075739032403212574055795686602260319041
7032406235170085879617892222789623703897374720000000000
0000000000000000000000000000000000000000000000000000000
Main>

```

```

factBig :: Integer -> Integer
factBig x
  | x == 0 = 1
  | otherwise = x * factBig (x-1)

```

Misturando Int e Integer

- Na verdade, o argumento do factorial podia ser Int. O resultado é que deve ser Integer:

```
factorial :: Int -> Integer
factorial x
  | x == 0 = 1
  | otherwise = x * factorial (x-1)
```

- Mas...

```
Main> factorial 100
ERROR file:f.hs:102 - Type error in application
*** Expression      : x * factorial (x - 1)
*** Term            : x
*** Type            : Int
*** Does not match  : Integer

Hugs>
```

Conversão de Int para Integer

- A função `fromIntegral` converte de `Int` para `Integer`:

```
factorial :: Int -> Integer
factorial x
  | x == 0 = 1
  | otherwise = fromIntegral x * factorial (x-1)
```

Em rigor, a função `fromIntegral` é mais geral do que isso: converte de qualquer tipo inteiro (`Int` ou `Integer`) para qualquer outro tipo numérico.

Exercícios

- Programe uma função para a potência de base inteira e expoente inteiro não negativo, baseada em multiplicações sucessivas, análoga à função `prod`.
- Idem, mas usando uma variante mais rápida, inspirada na função `prodFast`.
- Calcule o menor factorial com pelo menos 1000 algarismos.

Controlo

- Como se chama o tipo dos valores lógicos? De que operadores dispõe?
- Qual é o maior inteiro Int representável?
- Qual é o tipo dos inteiros de precisão arbitrária?
- Qual é o operador de potenciação?
- Qual é o maior número cujo factorial cabe nos inteiros Int?
- Quantos factoriais são números primos?
- Quantos factoriais são números ímpares?
- Qual é a função que converte de Int para Integer?
- Que faz a função div? E o operador `div`?
- E a função mod?

Na próxima aula...

- Estudaremos alguns algoritmos interessantes sobre números inteiros.

Fundamentos da Programação

Terceira aula:
Algoritmos sobre números inteiros:
máximo divisor comum

Nesta aula vamos...

- Estudar alguns algoritmos básicos que envolvem números inteiros, a propósito do máximo divisor comum.
- Aprender a programar genericamente, de maneira a que os nossos algoritmos dêem para números `Int` e para números `Integer`.
- Na próxima aula veremos alguns algoritmos sobre números primos.

Máximo divisor comum

- O que é o máximo divisor comum?
- Outro problema mais simples: qual é o maior divisor de um número x que é menor ou igual a um número dado y ?
- Bom, é y , se y for um divisor de x ; se não, é o maior divisor de x que é menor ou igual a $y-1$.
- Ora isto programa-se directamente.

Maior divisor

```
greatestDivisor :: Int -> Int -> Int
```

```
greatestDivisor x y
```

```
  | mod x y == 0 = y
```

```
  | otherwise = greatestDivisor x (y-1)
```

Ambos os argumentos
são números positivos.

Observe bem
esta técnica.

```
Main> greatestDivisor 100 99
50
Main> greatestDivisor 19 10
1
Main> greatestDivisor 201 201
201
Main> greatestDivisor 1000 80
50
Main> greatestDivisor 1024 500
256
Main> |
```

O maior número que
divide x e y e que é
menor ou igual a z

Máximo divisor comum, preliminares

- Generalizamos o esquema anterior para dois argumentos:

```
greatestCommonDivisor :: Int -> Int -> Int -> Int
greatestCommonDivisor x y z
  | mod x z == 0 && mod y z == 0 = z
  | otherwise = greatestCommonDivisor x y (z-1)
```

```
Main> greatestCommonDivisor 12345 6543 1000
3
Main> greatestCommonDivisor 36 30 30
6
Main> greatestCommonDivisor 1024 768 100
64
Main> |
```

Máximo divisor comum, versão 1

- Basta usar a função anterior, começando pelo mínimo entre x e y:

```
gcd0 :: Int -> Int -> Int
gcd0 x y = greatestCommonDivisor x y (min x y)
```

```
Main> gcd0 1000 640
40
Main> gcd0 123 321
3
Main> gcd0 35 72
1
Main> gcd0 800 800
800
Main> gcd0 1000 1001
```

Não conseguiremos
fazer melhor?

O mínimo de dois números

- A função min vem no prelúdio.
- Podíamos defini-la assim:

```
min' :: Int -> Int -> Int
min' x y = if x <= y then x else y
```

- Ou assim:

```
min'' :: Int -> Int -> Int
min'' x y
  | x <= y = x
  | otherwise = y
```

Algoritmo de Euclides

- Euclides inventou uma maneira melhor de calcular o máximo divisor comum, há mais de 2000 anos.
- Baseia-se na seguinte formulação:

$$\gcd(x, y) = \begin{cases} x, & \text{se } x = y \\ \gcd(x - y, y), & \text{se } x > y \\ \gcd(x, y - x), & \text{se } x < y \end{cases}$$

Função euclid

- Transcreve-se a formulação anterior para Haskell:

```
euclid :: Int -> Int -> Int
euclid x y
  | x < y = euclid x (y-x)
  | x > y = euclid (x-y) y
  | otherwise = x
```

Mais uma que não se esquece.

Alternativamente:

```
euclid' :: Int -> Int -> Int
euclid' x y =
  if x < y then euclid' x (y-x)
  else if x > y then euclid' (x-y) y
  else x
```

O algoritmo de Euclides é considerado o primeiro algoritmo não trivial inventado pelo espírito humano.

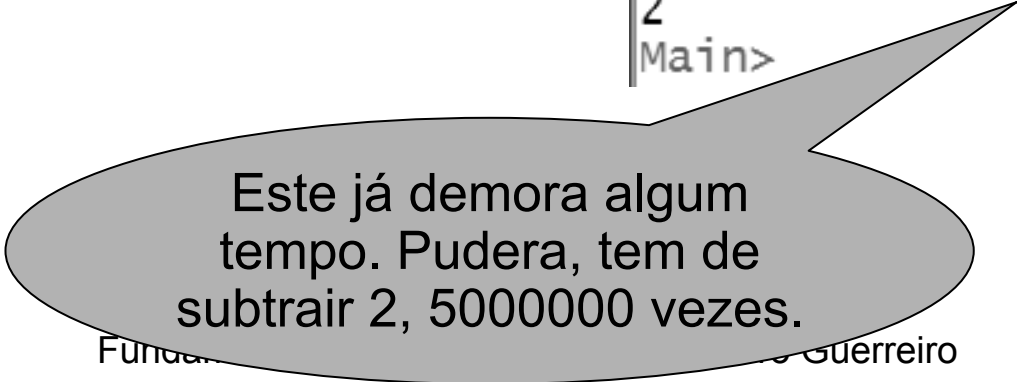
Experimentando

Casos “normais”:

```
Main> euclid 36 30
6
Main> euclid 19 31
1
Main> euclid 72 64
8
Main> euclid 10000 6225
25
Main>
```

Casos “espremidos”:

```
Main> euclid 2 10000
2
Main> euclid 2 100000
2
Main> euclid 2 1000000
2
Main> euclid 2 10000000
2
Main>
```



Este já demora algum tempo. Pudera, tem de subtrair 2, 5000000 vezes.

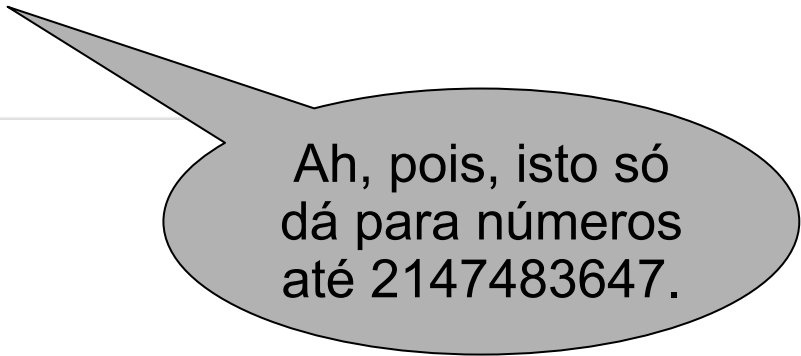
Mais experiências

- Com números maiores:

```
Main> euclid 545454 232323
10101
Main> euclid 54545454 23232323
1010101
Main> euclid 5454545454 2323232323

Program error: arithmetic overflow

Main>
```



Ah, pois, isto só dá para números até 2147483647.

euclid com Integer

```
euclidBig :: Integer -> Integer -> Integer
euclidBig x y
  | x < y = euclidBig x (y-x)
  | x > y = euclidBig (x-y) y
  | otherwise = x
```

```
Main> euclidBig 5454545454 2323232323
101010101
Main> euclidBig 54545454545454 23232323232323
1010101010101
Main> euclidBig 54545454545454545454 23232323232323232323
101010101010101010101
Main>
```

Muito bem, mas será que não podemos ter uma função única que dê para Int e para Integer?

euclid polimórfica

- Observe a nova função euclid, que substitui a anterior e a euclidBig:

```
euclid :: Integral a => a -> a -> a
```

```
euclid x y
```

```
  | x < y = euclid x (y-x)
```

```
  | x > y = euclid (x-y) y
```

```
  | otherwise = x
```

Isto significa que a função euclid se pode aplicar a quaisquer dois argumentos, desde que ambos sejam do mesmo tipo, e esse tipo pertença à classe Integral. O resultado é do mesmo tipo. Ora Int e Integer pertencem à classe de tipos Integral.

Mais experiências

- Com números grandes:

```
Main> euclid 10000000008 80000000001
{Interrupted!}

Main> euclid 868685656565 343436767767
1
Main> euclid 600000000000 200000000
200000000
Main> euclid 600000000000 40
{Interrupted!}

Main> |
```

- Por vezes demora tempo de mais: tivemos de interromper.
- Precisamos de uma função mais eficiente.

euclid, versão de combate

- A versão de combate, mais eficiente, tira partido da função mod para fazer de uma vez uma sequência de subtracções sucessivas:

```
gcd1 :: Integral a => a -> a -> a
gcd1 x y
  | y == 0 = x
  | otherwise = gcd1 y (mod x y)
```

Observando os tipos

- Se escrevemos `3::Int`, o Haskell interpreta esta expressão como valendo 3 e sendo do tipo `Int`.
- Se escrevemos `3::Integer`, idem, mas do tipo `Integer`.
- Se escrevemos 3 só, o Haskell atribui automaticamente o tipo que “der mais jeito”.

Aqui 50000000000 é grande de mais para ser do tipo `Int`.

```
Main> gcd1 40 50000000000
40
Main> gcd1 (40::Int) 50000000000

Program error: arithmetic overflow

Main> gcd (40::Int) (50::Integer)
ERROR - Type error in application
*** Expression      : gcd 40 50
*** Term            : 40
*** Type            : Int
*** Does not match  : Integer
```

OK, 40 aqui é do tipo `Integer`.

KO, o primeiro argumento é de tipo `Int` mas devia ser de tipo `Integer`.

Menor múltiplo comum

- A fórmula é:

$$lcm(x, y) = x \cdot y / gcd(x, y)$$

- Programamos assim, polimorficamente:

```
lcm :: Integral a => a -> a -> a
lcm x y = div x (gcd1 x y) * y
```

```
Main> lcm 60 96
480
Main> gcd1 480 60
60
Main> lcm 480 96
480
Main>
```

Exercícios

- Programe uma função para calcular o máximo divisor comum de três números.
- Idem, para o mínimo de três números.
- Torne polimórficas as diversas funções de hoje.
- Programe uma função booleana para verificar se dois números são primos entre si.
- Programe uma função divides, tal que divides x y calcula o número de vezes que é divisor de x. Por exemplo divides 72 3 vale 2; divides 78125 5 vale 7.

Controlo

- Qual foi o primeiro algoritmo criado pelo espírito humano? Para que serve?
- Qual é a diferença entre Integer e Integral?
- De cabeça: qual é o máximo divisor comum de 675 e 684?
- Qual é o erro quando usamos um Int fora dos limites?
- Para que serve o máximo divisor comum?
- Por que razão não estudámos uma função mínimo divisor comum?
- Qual é o máximo múltiplo comum de 27 e 35? :-S

Na próxima aula...

- Veremos as funções dos números primos.
- A partir de agora, sempre que for apropriado, as funções numéricas serão programadas polimorficamente.

Fundamentos da Programação

Quarta aula:
Números primos

Nesta aula vamos...

- Programar funções para números primos.
- Pelo caminho, estudar mais algumas técnicas fundamentais.
- Os números primos são objectos matemáticos muito interessantes, com utilidade prática em informática.

Números primos

- Em programação, é frequente precisarmos de saber se um número x é primo.
- Noutros casos, queremos saber qual é o x -ésimo número primo.

```
Main> isPrime 19
True
Main> isPrime 91
False
Main> isPrime 547
True
Main> isPrime 997
True
Main> isPrime 951
False

Main> prime 0
2
Main> prime 1
3
Main> prime 5
13
Main> prime 99
541
Main> prime 1000
7927
Main>
```

O que é um número primo?

- Por definição, um número inteiro positivo x é primo se for diferente de 1 e divisível apenas por x e por 1.
- Por conseguinte, x será primo se for maior do que 1 e se o seu maior divisor menor ou igual a $x-1$ for 1, ou, vendo ao contrário, se o seu menor divisor maior ou igual a 2 for x .

Menor divisor maior que

- É parecido com o maior divisor menor que:

```
leastDivisor :: Int -> Int -> Int
leastDivisor x y
  | mod x y == 0 = y
  | otherwise = leastDivisor x (y+1)
```

Recorde,
da aula
anterior

```
greatestDivisor :: Int -> Int -> Int
greatestDivisor x y
  | mod x y == 0 = y
  | otherwise = greatestDivisor x (y-1)
```

isPrime

```
isPrime :: Int -> Bool
-- pre: x >= 2
isPrime x = leastDivisor x 2 == x
```

O comentário é para não nos esquecermos de que o argumento tem de ser maior ou igual a 2.

- Note bem: isto não é propriamente um algoritmo, no sentido do algoritmo de Euclides, por exemplo; é apenas a aplicação directa da definição.

Melhorando

- Na verdade, não devia ser preciso procurar um divisor de x entre 2 e x , mas apenas entre 2 e a raiz quadrada de x , pois, se houver um divisor então há dois (porventura iguais) e o menor deles é menor ou igual à raiz de x .
- Quer dizer, entre 2 e x , há sempre pelo menos um divisor de x , nem que seja só o próprio x , mas entre 2 e a raiz de x pode não haver, caso em que x é primo.
- Por outro lado, se não houver um divisor entre 2 e a raiz quadrada de x , a função que procura o divisor devolve o quê?

Talvez um número inteiro

- Problema: calcular o menor divisor de x no intervalo de y até z .
- Talvez haja, talvez não:

Observe esta técnica.

```
leastDivisorFromTo :: Int -> Int -> Int -> Maybe Int
```

```
leastDivisorFromTo x y z
```

```
  | y > z = Nothing
```

Intervalo vazio.
Não há nada.

```
  | mod x y == 0 = Just y
```

O resto é zero. O
resultado é Just y.

```
  | otherwise = leastDivisorFromTo x (y+1) z
```

```
Main> leastDivisorFromTo 40 11 50
Just 20
Main> leastDivisorFromTo 40 11 15
Nothing
Main>
```

isPrimeBetter

- Faz o mesmo que isPrime, mas com menos esforço:

```
isPrimeBetter :: Int -> Bool
isPrimeBetter x =
    leastDivisorFromTo x 2 (sqrtInt x) == Nothing
```

Falta
programar a
função
sqrtInt.

Raiz quadrada

- A função `sqrt` do prelúdio calcula a raiz quadrada de números reais.
- Aqui queremos a raiz quadrada de inteiros.
- A função seguinte calcula o maior número inteiro cujo quadrado é menor ou igual ao argumento:

```
sqrtInt :: Integral a => a -> a  
sqrtInt x = floor (sqrt (fromIntegral x))
```

Repare: temos de usar `fromIntegral x` porque a função `sqrt` está à espera de um número `Double` e `x` é de um tipo `Integral` (`Int` ou `Integer`).

Experimentando a raiz

```
Main> sqrt 1000
31.6227766016838
Main> sqrtInt 1000
31
Main> sqrt 1024
32.0
Main> sqrtInt 1024
32
Main> 123456789 * 123456789
15241578750190521
Main> sqrt 15241578750190521
123456789.0
Main> sqrtInt 15241578750190521
123456789
Main> 15241578750190521 * 15241578750190521
232305722798259244150093798251441
Main> sqrt 232305722798259244150093798251441
1.52415787501905e+016
Main> sqrtInt 232305722798259244150093798251441
15241578750190520
Main> 232305722798259244150093798251441 * 232305722798259244150093798251441
53965948844821664748141453212125737955899777414752273389058576481
Main> sqrt 53965948844821664748141453212125737955899777414752273389058576481
2.32305722798259e+032
Main> sqrtInt 53965948844821664748141453212125737955899777414752273389058576481
232305722798259242383383568842752
Main> |
```

Não funciona para números grandes, pois os reais têm precisão finita. Por isso, ao calcular a raiz usando a função sqrt para números grandes, perdemos precisão, irremediavelmente. ☹

Voltaremos a esta questão mais tarde.

O próximo primo

- Qual é o próximo primo, isto é, dado um número x , qual o menor número primo maior ou igual a x ?

```
nextPrime :: Int -> Int
nextPrime x
  | isPrimeBetter x = x
  | otherwise = nextPrime (x+1)
```

Esta função calcula, porque o conjunto dos números primos é infinito: dado um número x , há sempre um número primo maior ou igual a x . (De facto, há infinitos...)

O próximo primo, melhor

- Em geral, podemos avançar de 2 em 2, poupando metade do trabalho. Observe:

```
nextPrime :: Int -> Int
nextPrime x
  | x <= 2 = 2
  | even x = nextPrime' (x+1)
  | otherwise = nextPrime' x
```

```
nextPrime' :: Int -> Int
nextPrime' x
  | isPrimeBetter x = x
  | otherwise = nextPrime' (x+2)
```

Esta função
substitui a
anterior.

Esta também
daria, mas
trabalha mais.

```
nextPrime :: Int -> Int
nextPrime x
  | x <= 2 = 2
  | even x = nextPrime (x+1)
  | isPrimeBetter x = x
  | otherwise = nextPrime (x+2)
```

O primo de ordem x

- Queremos saber qual é o x -ésimo primo, começando a contar por zero: o zero-ésimo primo é 2, o um-ésimo é 3, ..., o 23-ésimo é 89.

```
prime :: Int -> Int
```

```
prime x
```

```
    | x == 0 = 2
```

```
    | x == 1 = 3
```

```
    otherwise = nextPrime' (prime (x-1) + 2)
```

```
Main> prime 0
```

```
2
```

```
Main> prime 0
```

```
2
```

```
Main> prime 1
```

```
3
```

```
Main> prime 5
```

```
13
```

```
Main> prime 23
```

```
89
```

```
Main> prime 100
```

```
547
```

```
Main> prime 500
```

```
3581
```

```
Main> |
```

- Mas não dá para argumentos grandes ☹
Temos de voltar a isto mais tarde.

```
Main> prime 4000
```

```
ERROR - C stack overflow
```

```
Main>
```

Exercício ilustrativo

- Um super-primo é um número primo tal que se apagarmos qualquer sufixo, o número resultante também é primo. Por exemplo, 7193 é um super-primo. (Repare, 7193, 719, 71 e 7 são números primos.).
- Queremos programar uma função que calcula o primeiro super-primo maior ou igual a um número dado.

Resolução (1), os prefixos

- “Apagamos” os sucessivos sufixos, obtendo os sucessivos prefixos, dividindo sucessivamente por 10:

```
isSuperPrime :: Int -> Bool
isSuperPrime x
  | x < 10 = elem x [2, 3, 5, 7]
  | otherwise = isPrime x && isSuperPrime (div x 10)
```

Dá True se x
for um dos
valores da lista.

Resolução (2), o próximo

```
nextSuperPrime :: Int -> Int
nextSuperPrime x
  | x < 2 = 2
  | x == 2 = 3
  | even x = nextSuperPrime (x+1)
  | isSuperPrime x = x
  | otherwise = nextSuperPrime (x+2)
```

Uma versão mais explícita, análoga a nextPrime, e outra mais compacta, tirando partido de nextPrime.

```
nextSuperPrime' :: Int -> Int
nextSuperPrime' x
  | isSuperPrime x = x
  | otherwise = nextSuperPrime' (nextPrime (x+1))
```

```
Main> nextSuperPrime 100
233
Main> nextSuperPrime' 100
233
Main> nextSuperPrime 500
593
Main> nextSuperPrime' 500
593
Main> nextSuperPrime 6000
7193
Main> nextSuperPrime' 6000
7193
Main>
```

Exercícios

- Dois números primos x e y dizem-se primos gémeos se x e y são primos e $y = x+2$. Programe uma função para verificar se dois números são primos gémeos.
- Programe uma função para calcular o primeiro par de primos gémeos em que ambos são maiores ou iguais a um número dado. A função devolve apenas o primeiro dos primos, já que o segundo é esse mais 2.
- Programe uma função para verificar se todos os algarismos de um número são números primos (2, 3, 5 ou 7)?
- Programe uma função para calcular o número de divisores próprios de um número. (Um divisor próprio de x é um divisor de x diferente de 1 e de x .)
- Programe uma função que calcule o primeiro número com mais do que x divisores no intervalo de y a z , ou Nothing, se não houver nenhum nessas condições.

Controlo

- Quantos números primos há?
- Como se chama função do prelúdio que calcula a raiz quadrada.
- Qual é o maior primo menor do que 1000.
- Qual é o 100-ésimo número primo? E o centésimo? (Qual é o 1-ésimo? Qual é o primeiro'?)
- Usámos o tipo Maybe Int para quê?

Na próxima aula

- Começaremos a estudar listas.
- De facto, até agora, cada função manipulava um pequeno número de valores, representados pelos argumentos.
- Frequentemente, temos muitos valores para processar. Em Haskell, esses valores virão em listas.

Fundamentos da Programação

Quinta aula: Listas

Nesta aula vamos...

- Começar a estudar as listas.
- As listas são as estruturas de dados que servem para agrupar sequencialmente um número indeterminado de valores de um mesmo tipo.
- As listas são muito importantes. Temos de aprender a programar com elas.

Exemplos de problemas com listas

- Calcular a nota final das auto-avaliações, dadas as notas das auto-avaliações realizadas e o número de auto-avaliações propostas.
- Calcular a média final do curso, dadas as notas de cada cadeira e os créditos respectivos.
- Calcular a nota da escala europeia de comparabilidade, dada a nota e a lista das notas dos diplomados nos últimos três anos.

Estes são inspirados pelo problema da nota.

Mais exemplos

- Calcular a lista dos divisores de um número.
- Calcular a lista dos factores primos de um número.
- Factorizar um número.
- Construir a lista com os x primeiros números primos.
- Calcular a lista dos valores do maior argumento nos sucessivos passos do algoritmo de Euclides.

Estes são relacionados com os exemplos das aulas anteriores.

Listas

- As listas denotam-se entre parêntesis rectos, com os elementos, todos do mesmo tipo, separados por vírgulas:

```
Main> [2,3,5,7,11,13,17,19,23,29]
```

```
[2,3,5,7,11,13,17,19,23,29]
```

```
Main> [16.4,17.1,19.3,11.0]
```

```
[16.4,17.1,19.3,11.0]
```

```
Main> [True,True,False,True,False]
```

```
[True,True,False,True,False]
```

```
Main> [[1,2],[8,3]]
```

```
[[1,2],[8,3]]
```

```
Main> [2,7,3.14]
```

```
[2.0,7.0,3.14]
```

```
Main> []
```

```
[]
```

```
Main>
```

Lista
vazia.

```
Main> [2::Int,7,3.14]
```

```
ERROR - Cannot infer instance
```

```
*** Instance    : Fractional Int
```

```
*** Expression : [2,7,3.14]
```

```
Main> [1, True]
```

```
ERROR - Cannot infer instance
```

```
*** Instance    : Num Bool
```

```
*** Expression : [1,True]
```

```
Main>
```

Tipos das listas

- As listas de números Int são de tipo [Int].
- E analogamente para as outras.

```
Main> :type [6,1,9,0,4]
[6,1,9,0,4] :: Num a => [a]
Main> :type [6::Int, 1::Int]
[6,1] :: [Int]
Main> :type [True,True,False]
[True,True,False] :: [Bool]
Main> :type [2::Int, 3, 4]
[2,3,4] :: [Int]
Main> :type [4,8,6::Double]
[4,8,6] :: [Double]
Main> :type []
[] :: [a]
Main> |
```


Operações básicas

- null, dá True se a lista for vazia.
- head, dá o primeiro elemento da lista.
- tail, dá a lista que se obtém removendo o primeiro elemento
- Operador :, constrói uma lista dados um valor, que constituirá o primeiro elemento da lista construída, e uma lista, que constituirá o resto da lista construída.

null

```
Main> null []  
True  
Main> null [2,4,6,8,10]  
False  
Main> null [False]  
False  
Main> null [[]]  
False  
Main> null [[[[[]]]]]  
False  
Main> null [0]  
False  
Main> null [4.5,6.6,9.0]  
False  
Main> |
```

head

```
Main> head [7,4,9,1]
7
Main> head [False]
False
Main> head [1.1,2.2,3.3]
1.1
Main> head [[]]
[]
Main> head [3]
3
Main> head [10,9,8,7,6,5,4,3,2,1]
10
Main> head [True,True,True,False]
True
Main> head []
```

As listas vazias
não têm cabeça

```
Program error: pattern match failure: head []
```

```
Main> |
```

tail

```
Main> tail [8,5,4,3]
[5,4,3]
Main> tail [6]
[]
Main> tail [False,True]
[True]
Main> tail [4,5,6,7,8,9,10,11]
[5,6,7,8,9,10,11]
Main> tail [9.9,7.1,2.3,4.5]
[7.1,2.3,4.5]
Main> tail []
```

As listas vazias
não têm cauda.

Program error: pattern match failure: tail []

Main>

Operador :

```
Main> 4 : [8,3]
[4,8,3]
Main> 6 : []
[6]
Main> [] : []
[[]]
Main> True : [False,False,False,True,False]
[True,False,False,False,True,False]
Main> [3,5] : [[2],[8,2,3]]
[[3,5],[2],[8,2,3]]
Main> 6.2 : [4,5.8,4.6]
[6.2,4.0,5.8,4.6]
Main> 3 : 4 : [5]
[3,4,5]
Main> 3 : 4 : 5 : []
[3,4,5]
Main> 7 : 2 : [3,5,1]
[7,2,3,5,1]
Main> 9 : 7 : 5 : 3 : []
[9,7,5,3]
Main>
```

Outras operações típicas

- `length xs`, número de elementos de `xs`.
- `elem x ys`, `x` é um dos elementos de `ys`?
- `sum xs`, soma dos elementos de `xs`.
- `product xs`, produto dos elementos de `xs`.
- `last xs`, o último elemento de `xs`.
- `replicate x y`, uma lista com `x` elementos todos iguais a `y`.
- `reverse xs`, uma lista com os mesmo elementos que `xs` mas pela ordem inversa.
- `minimum xs`, o menor elemento de `xs`.
- `maximum xs`, o maior elemento de `xs`.

Estas vêm todas no prelúdio.

Programando a soma

- Não precisamos de programar a soma, mas é um belo exercício:

```
sumList :: [Int] -> Int
sumList xs
  | null xs = 0
  | otherwise = head xs + sumList (tail xs)
```

```
Main> sumList [4,8,10]
22
Main> sumList []
0
Main> sumList [100..105]
615
Main>
```

Programando o comprimento

- Também é muito interessante:

```
lengthList :: [Int] -> Int
```

```
lengthList xs
```

```
  | null xs = 0
```

```
  | otherwise = 1 + length (tail xs)
```

Em relação a sumList, soma-se 1 no otherwise, em vez de head xs.

```
Main> lengthList [9,4,8,1,2]
```

```
5
```

```
Main> lengthList []
```

```
0
```

```
Main> lengthList [2..999]
```

```
998
```

```
Main> |
```


Programando a função elem

- Esta é um pouco diferente das anteriores:

```
elemList :: Int -> [Int] -> Bool
```

```
elemList x ys
```

```
  | null ys = False
```

```
  | otherwise = x == head ys || elemList x (tail ys)
```

```
Main> elemList 3 [9,6,5,3,7]
```

```
True
```

```
Main> elemList 6 [1..5]
```

```
False
```

```
Main> elemList 10 []
```

```
False
```

```
Main>
```

- Alternativamente:

```
elemList :: Int -> [Int] -> Bool
```

```
elemList x ys = not (null ys) && (x == head ys || elemList x (tail ys))
```

O mínimo da lista

- Sem palavras:

```
minimumList :: [Int] -> Int
minimumList xs
  | null t = h
  | otherwise = min h (minimumList t)
  where
    h = head xs
    t = tail xs
```

- Ou:

```
minimumList :: [Int] -> Int
minimumList' xs
  | null (tail xs) = head xs
  | otherwise = min (head xs) (minimumList (tail xs))
```

Outro exemplo: a lista dos factores

- Se x for primo, então a lista dos factores de x é a lista unitária $[x]$.
- Caso contrário, é a lista cujo primeiro elemento é o menor divisor de x , seguida da lista dos divisores do quociente de x por esse menor divisor.

É mais fácil
programar do
que explicar.

```
Main> factors 20
[2,2,5]
Main> factors 336
[2,2,2,2,3,7]
Main> factors 22540
[2,2,5,7,7,23]
Main> factors (2^20)
[2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2]
Main> factors 37
[37]
Main>
```

```
factors :: Int -> [Int]
factors x
  | isPrimeBetter x = [x]
  | otherwise = z : factors (div x z)
  where
    z = leastDivisor x 2
```

Exercícios

- Programe a função que dá o último elemento de uma lista não vazia.
- Programe a função que dá o x -ésimo elemento de uma lista com pelo menos $x+1$ elementos.
- Programe a função `replicateList`, análoga à `replicate` do prelúdio.
- Programe a função `maximumList`, que calcula o máximo de uma lista não vazia.
- Programe uma função para fazer a conjunção lógica dos valores presentes numa lista de booleanos.
- Idem, para a disjunção.
- Programe uma função para produzir a lista dos sucessores dos elementos de outra lista. Por exemplo, `f [8,10,3]` vale `[9,11,4]`.

Controlo

- Quais são as funções básicas das listas.
- Qual é o tipo das listas de números inteiros Int.
- A lista vazia é de que tipo?
- Quantas funções programámos hoje?

Na próxima aula...

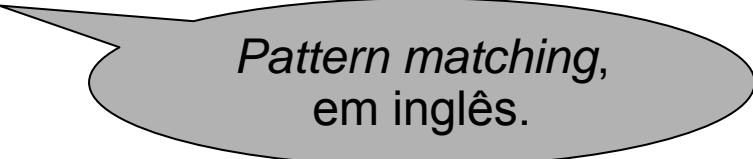
- Continuaremos a estudar as listas.
- Aprenderemos as técnicas do encontro de padrões na definição de funções.

Fundamentos da Programação

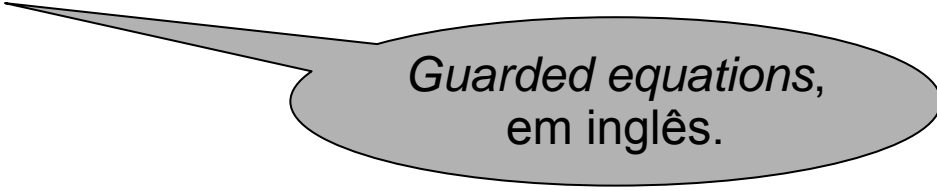
Sexta aula:
Encontro de padrões

Nesta aula vamos...

- Aprender uma nova técnica para definir funções em Haskell: *encontro de padrões*.
- Daqui para a frente, será a técnica mais usada, em combinação com a das *equações guardadas*, que temos usado.



Pattern matching,
em inglês.



Guarded equations,
em inglês.

Primeiro exemplo: sumList

- Antes, programávamos assim, com equações guardadas:

```
sumList :: [Int] -> Int
sumList xs
  | null xs = 0
  | otherwise = head xs + sumList (tail xs)
```

- A partir de agora, usaremos encontro de padrões, quando for mais expressivo:

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

Os parêntesis são precisos em `sumList (x:xs)` porque a chamada de função tem mais prioridade do que todos os outros operadores.

Comprimento

- Assim:

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

- Ou usando um *diabrete*, já que o valor encontrado não é usado na definição:

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

Usamos o termo *diabrete* como tradução do inglês *wildcard*.

Diabrete

- A lista tem pelo menos três elementos?

```
longerThan3 :: [Int] -> Bool  
longerThan3 (_:_:_:_) = True  
longerThan3 _ = False
```

```
longerThan3' :: [Int] -> Bool  
longerThan3' x = length x >= 3
```

- A lista tem exactamente três elementos?

```
isTriplet :: [Int] -> Bool  
isTriplet [_,_,_] = True  
isTriplet _ = False
```

Note bem: estas duas fazem o mesmo mas trabalham mais, porque têm de calcular o comprimento.

```
isTriplet' :: [Int] -> Bool  
isTriplet' x = length x == 3
```

Diabrete, mais exemplos

- A lista começa por zero?

```
startsByZero :: [Int] -> Bool
startsByZero (0:_) = True
startsByZero _ = False
```

Mas não esqueça a outra maneira de programar:

```
startsByZero' :: [Int] -> Bool
startsByZero' xs = head xs == 0
```

- Quantas listas não vazias há numa lista de listas?

```
countNonEmpty :: [[Int]] -> Int
countNonEmpty [] = 0
countNonEmpty ((_:_) : xss) = 1 + countNonEmpty xss
countNonEmpty (_ : xss) = countNonEmpty xss
```

```
countNonEmpty' :: [[Int]] -> Int
countNonEmpty' [] = 0
countNonEmpty' (xs:xss) =
  fromEnum (not (null xs)) +
  countNonEmpty' xss
```

Mais exemplos de padrões

- elemList:

```
elemList :: Int -> [Int] -> Bool
```

```
elemList _ [] = False
```

```
elemList x (y:ys) = x == y || elemList x ys
```

```
Main> elemList 5 [0..10]
True
Main> elemList 5 [9..10]
False
Main> elemList 0 []
False
Main> elemList 8 [8]
True
Main>
```

- minimumList:

```
minimumList :: [Int] -> Int
```

```
minimumList [x] = x
```

```
minimumList (x:xs) = min x (minimumList xs)
```

```
Main> minimumList [3,6,8,2,9,8]
2
Main> minimumList [7,7,7,7,7]
7
Main> minimumList [23]
23
Main> minimumList []
```

```
Program error: pattern match failure:
minimumList []
```

```
Main>
```

error

- Querendo, podemos nós próprios detectar as situações de erro e documentá-las na definição da função, usando a função `error`:

```
minimumList :: [Int] -> Int
minimumList [] = error "minimumList: empty list"
minimumList [x] = x
minimumList (x:xs) = min x (minimumList xs)
```

```
Main> minimumList []
```

```
Program error: minimumList: empty list
```

```
Main> |
```

head, tail, cons, se fosse preciso

- **headList:**

```
headList :: [Int] -> Int  
headList [] = error "headList: empty list"  
headList (x:_) = x
```

- **tailList:**

```
tailList :: [Int] -> [Int]  
tailList [] = error "tailList: empty list"  
tailList (_:xs) = xs
```

- **cons:**

```
cons :: Int -> [Int] -> [Int]  
cons x ys = x:ys
```

cons ao contrário?

- O operador (:) constrói uma lista com a cabeça dada e com a cauda dada.
- De certa forma, acrescenta um novo elemento no início da lista.
- E como faríamos para acrescentar o novo elemento no fim da lista?

```
snoc :: Int -> [Int] -> [Int]
```

```
snoc x ys = ys:x
```

```
Main> :reload  
ERROR file:pattern_matching.hs:182 - Type error  
in application  
*** Expression      : ys : x  
*** Term            : x  
*** Type            : Int  
*** Does not match  : [a]
```


snoc

- Acrescentar um elemento no fim da lista não é uma função primitiva.
- Programemo-la, como exercício:

```
snoc :: Int -> [Int] -> [Int]
snoc x [] = [x]
snoc x (y:ys) = y : snoc x ys
```

```
Main> snoc 4 [9,2,3]
[9,2,3,4]
Main> snoc 88 []
[88]
Main> snoc 1000 [1..5]
[1,2,3,4,5,1000]
Main>
```

- Observe os cálculos:

```
snoc 4 [9,2,3] =
9 : snoc 4 [2,3] =
9 : (2 : snoc 4 [3]) =
9 : (2 : (3 : snoc 4 [])) =
9 : (2 : (3 : (4 : []))) =
[9,2,3,4]
```

Note bem: por exemplo [1,2,3] é, tecnicamente, apenas uma maneira diferente de escrever 1:(2:(3:[])), ou, 1:2:3:[], sem parêntesis, porque (:) associa à direita. Logo [4] é 4:[] e 9:2:3:4:[] é [9,2,3,4]

Concatenação

- Programemos agora a concatenação de listas, com a função `append`.
- Aliás, dispondo da concatenação, a função `snoc` programa-se `snoc x ys = append ys [x]`, e pronto.

```
append :: [Int] -> [Int] -> [Int]
append [] ys = ys
append (x:xs) ys = x : append xs ys
```

```
Main> append [2,4,6] [1,3]
[2,4,6,1,3]
Main> append [] [8,9,10]
[8,9,10]
Main> append [2..6] []
[2,3,4,5,6]
Main> append [1..5] [1000]
[1,2,3,4,5,1000]
Main>
```

Isto é um clássico.
Não se esqueça!

Cálculo da concatenação

- Observe o exemplo

```
append [2,4,6][1,3] =  
2 : append[4,6][1,3] =  
2 : (4 : append [6][1,3]) =  
2 : (4 : (6 : (append [][1,3]))) =  
2 : (4 : (6 : [1,3])) =  
[2,4,6,1,3]
```

- De certa forma, a primeira lista é desconstruída e depois reconstruída do fim para o princípio, já colada à segunda lista.
- Isto é muito interessante, mas no prelúdio já temos o operador (++) que faz a concatenação de listas 😊.

Padrões de inteiros

- Também há padrões de inteiros.
Observe o factorial:

```
factorial :: Int -> Integer
```

```
factorial 0 = 1
```

```
factorial (x+1) = fromIntegral (x+1) * factorial x
```

```
Main> factorial 20  
2432902008176640000  
Main> factorial 1  
1  
Main> factorial 0  
1  
Main> factorial (-1)
```

Note bem: os padrões $x+k$ só encontram números maiores ou iguais a k . Assim, na definição acima, os dois padrões não se sobrepõem, e os números negativos não são encontrados.

```
Program error: pattern match failure: factorial (-1)
```

```
Main>
```

Potência

- Eis a potência, na sua versão linear, com controlo dos expoentes negativos, por encontro de padrões:

```
power :: Integer -> Int -> Integer
power 0 0 = error "power: power 0 0 is undefined"
power 0 1 = 0
power _ 0 = 1
power x (y+1) = x * power x y
power _ _ = error "power: negative exponent"
```

```
Main> power 3 4
81
Main> power 5 0
1
Main> power 0 0
0
Main> power 0 7
0
Main> power (-2) 11
-2048
Main> power 6 (-3)
Program error: power: negative exponent
Program error: power: power 0 0 is undefined
```

Inversão

- Para inverter uma lista, há a função `reverse`, no prelúdio.
- Programemos, para apreciar bem:

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

```
Main> rev [4..10]
[10,9,8,7,6,5,4]
Main> rev [6,2,9,7]
[7,9,2,6]
Main> rev [4]
[4]
Main> rev []
[]
Main>
```

```
rev :: [Int] -> [Int]
rev [] = []
rev (x:xs) = append (rev xs) [x]
```

Variante, com `append`!

Cálculo da inversão

```

rev [6,2,9,7] =
append (rev [2,9,7]) [6] =
append (append (rev [9,7]) [2]) [6] =
append (append (append (rev [7]) [9]) [2]) [6] =
append (append (append (append (rev []) [7]) [9]) [2]) [6] =
append (append (append (append [] [7]) [9]) [2]) [6] =
append (append (append [7] [9]) [2]) [6] =
append (append (7: append [] [9]) [2]) [6] =
append (append (7: [9]) [2]) [6] =
append (append [7,9] [2]) [6] =
append (7: append [9] [2]) [6] =
append (7: 9: append [] [2]) [6] =
append (7: 9: [2]) [6] =
append [7,9,2] [6] =
7:append [9,2][6] =
7:9:append[2][6] =
7:9:2:append[] [6] =
7:9:2:[6] =
[7,9,2,6]
    
```

A inversão trabalha muito ☹. Por cada elemento da lista a inverter há um append. Ao fazer um append de uma lista com n elementos fazem-se n cons. Logo, inverter uma lista com n elementos fazem-se $n+(n-1)+ \dots + 1$ cons, isto é, $(n+1)*n/2 \approx n^2/2$ cons.

Exercícios

- Programe uma variante da função append, baseada na função snoc, que desconstrói a segunda lista acrescentado cada elemento no fim da primeira.
- Programe uma função que concatena todas as listas de uma lista de listas.
- Programe uma função que dada uma lista não vazia constrói outra em que o primeiro elemento da primeira passa para último lugar.
- Reprograme a função da multiplicação de inteiros usando padrões.
- Programe uma função dup que substitua cada elemento por duas ocorrências desse elemento: $\text{dup } [3,5,2] = [3,3,5,5,2,2]$.

Controlo

- Qual é mais eficiente: cons ou snoc?
- Qual é o operador do prelúdio para a concatenação d listas?
- Inverter duas listas e depois concatená-las é o mesmo que concatenar as duas listas e inverter o resultado?
- Interprete a igualdade `head xs : tail xs == xs`, qualquer que seja a lista não vazia `xs`.
- Interprete a igualdade `reverse (reverse xs) == xs`, qualquer que seja a lista `xs`.
- Quantos cons se fazem para inverter a lista `[1..1000]`?

Na próxima aula

- Estudaremos listas de pares.
- Aprenderemos a ordenar listas.

Fundamentos da Programação

Sétima aula:
Pares e listas de pares

Nesta aula vamos...

- Conhecer os pares, em Haskell.
- Pares são tuplos com dois elementos.
- Um tuplo é uma sequência finita de componentes, cada um com o seu tipo (mas não é preciso que os tipos sejam todos diferentes, claro).
- Veremos também alguns algoritmos com listas de pares.

Qual foi a cidade mais quente?

- Registámos a temperatura máxima de um conjunto de cidades. Qual foi a cidade mais quente?
- Antes disso: qual foi a maior temperatura observada?
- Cada registo é um par, por exemplo ("aveiro", 13.6).
- A lista dos registos é uma lista de pares:
[("aveiro", 13.6), ("porto", 12.9), ("lisboa", 15.1), ("faro", 17.2)].

Os tipos

- O nome da cidade é de tipo String.
- A temperatura é de tipo Double.
- O tipo de cada registo é par String, Double: (String, Double)
- A lista é uma lista de pares (String, Double): [(String, Double)]
- A função da temperatura máxima tem a seguinte assinatura:

```
maxTemperature :: [(String, Double)] -> Double
```

`fst, snd`

- O valor do primeiro elemento do par `p` é dado por `fst p`, e o do segundo por `snd p`.
- `fst` e `snd` são funções do prelúdio, que dão para pares seja do que for.
- Exemplo, a distância entre dois pontos representados pelos seus pares de coordenadas:

```
distance :: (Double, Double) -> (Double, Double) -> Double
distance p q = sqrt ((fst p - fst q)^2 + (snd p - snd q)^2)
```

```
Main> distance (0,0)(3,4)
5.0
Main> distance (110.0,110.0)(210.0,210.0)
141.42135623731
Main> distance (0.1,0.6)(-1.5,0.6)
1.6
Main>
```

Padrões

- Frequentemente, dispensamos as funções `fst` e `snd`, preferindo os padrões de pares.
- Observe a seguinte formulação da função da distância:

```
distance' :: (Double, Double) -> (Double, Double) -> Double  
distance' (x1, y1)(x2, y2) = sqrt ((x1 - x2)^2 + (y1 - y2)^2)
```

```
Main> distance' (3,4)(0,0)  
5.0  
Main> distance' (-2.3,5.0)(-2.3,5.0)  
0.0  
Main> |
```


Outra questão: a lista de observações numa cidade

- Dada a cidade, calcular a lista das temperaturas observadas nessa cidade:

Nota: esta questão é mais geral do que necessário, pois no contexto do presente exemplo, cada cidade só tem uma temperatura associada.

```
find :: String -> [(String, Double)] -> [Double]
```

```
find _ [] = []
```

```
find x ((a,b):ts)
```

```
  | x == a = b : z
```

```
  | otherwise = z
```

```
  where
```

```
    z = find x ts
```

```
Main> find "aa" []
```

```
[]
```

```
Main> find "aa" [("rr",7),("aa",8),("ss",3),  
                ("rr",7),("aa",48),("ww",3)]
```

```
[8.0,48.0]
```

```
Main> find "zz" [("rr",7),("aa",8),("ss",3),  
                ("rr",7),("aa",48),("ww",3)]
```

```
[]
```

```
Main> find "ss" [("rr",7),("aa",8),("ss",3),  
                ("rr",7),("aa",48),("ww",3)]
```

```
[3.0]
```

```
Main>
```

A temperatura máxima

- É como o máximo da lista de inteiros, mas só observando o segundo elemento de cada par:

```
maxTemperature :: [(String,Double)] -> Double
maxTemperature [] = error "maxTemperature: empty list"
maxTemperature [(_,t)] = t
maxTemperature ((_,t):xs) = max t (maxTemperature xs)
```

```
Main> maxTemperature [("lisboa",15.3),("setubal",17.7),
("evora",14.8),("castelo branco",12.2),("faro",18.2),
("viseu",17.7),("porto",11.7),("aveiro",14.2),
("beja",18.1),("braganca",9.6)]
18.2
```

```
Main> maxTemperature [("lisboa",15.3)]
15.3
Main> maxTemperature []
```

```
Program error: maxTemperature: empty list
```

09-01-:

```
Main>
```

Listas "constantes"

- Para evitar ter de escrever longas listas quando queremos experimentar os nossos programas, usamos listas constantes:

```
yesterday1 :: [(String,Double)]
yesterday1 = [("lisboa", 15.3), ("setubal", 17.7), ("evora", 14.8), ("castelo branco", 12.2),
              ("faro", 18.2), ("viseu", 17.7), ("porto", 11.7), ("aveiro", 14.2), ("beja", 18.1),
              ("braganca", 9.6)]
yesterday2 :: [(String,Double)]
yesterday2 = [("lisboa", 15.3), ("setubal", 17.7), ("evora", 14.8), ("castelo branco", 12.2),
              ("faro", 18.2), ("viseu", 17.7), ("porto", 11.7), ("aveiro", 14.2), ("beja", 18.2),
              ("braganca", 9.6)]
```

```
Main> maxTemperature yesterday1
18.2
Main> |
```

zip

- A função zip emparelha duas listas, criando uma lista de pares.
- Vem no prelúdio, e dá para listas de quaisquer tipos.
- Programemos, para listas de inteiros:

```
zipList :: [Int] -> [Int] -> [(Int, Int)]  
zipList [] _ = []  
zipList _ [] = []  
zipList (x:xs) (y:ys) = (x, y) : zipList xs ys
```

```
Main> zipList [1..5] [4..12]  
[(1,4),(2,5),(3,6),(4,7),(5,8)]  
Main> zipList [1..1000] []  
[]
```

unzip

- A função unzip desemparelha uma listas de pares, criando duas listas.
- Tal com o função zip, vem no prelúdio e dá para listas de pares quaisquer.
- Dois estilos de programação:

```
unzipList :: [(Int, Int)] -> ([Int], [Int])
unzipList [] = ([], [])
unzipList (x:xs) = (fst x : fst z, snd x : snd z)
  where
    z = unzipList xs
```

```
unzipList' :: [(Int, Int)] -> ([Int], [Int])
unzipList' [] = ([], [])
unzipList' ((a,b):xs) = (a : as, b : bs)
  where
    (as,bs) = unzipList' xs
```

```
Main> unzipList [(1,4),(2,5),(3,6)]
([1,2,3],[4,5,6])
Main> unzipList []
([],[])
Main> |
```

A temperatura máxima, com unzip

- Recorrendo à função unzip, a temperatura máxima calcula-se "sem esforço":

```
maxTemperature' :: [(String,Double)] -> Double  
maxTemperature' xs = maximum (snd (unzip xs))
```

```
Main> maxTemperature' yesterday1
```

```
18.2
```

```
Main> maxTemperature' []
```

```
Program error: pattern match failure: foldl1 max []
```

```
Main>
```

Não previmos o caso da lista vazia, e por isso dá erro no cálculo da função maximum.

As cidades mais quentes

- Agora que já sabemos a temperatura máxima, calculemos a lista das cidades onde essa temperatura máxima ocorreu.
- Começemos por calcular a lista de cidades "mais quentes que"

```
hotterThan :: Double -> [(String,Double)] -> [String]
```

```
hotterThan _ [] = []
```

```
hotterThan x ((a,b):ys)
```

```
  | x <= b = a : h
```

```
  | otherwise = h
```

```
  where
```

```
    h = hotterThan x ys
```

```
Main> hotterThan 16.0 yesterday1  
["setuba", "faro", "viseu", "beja"]  
Main> hotterThan 14.5 yesterday1  
["lisboa", "setuba", "evora", "faro", "viseu", "beja"]  
Main> |
```

E se quiséssemos a lista por ordem alfabética?

As mais quentes de todas

- São aquelas cuja temperatura é maior ou igual à temperatura máxima calculada:

```
hottest :: [(String,Double)] -> [String]
hottest xs = hotterThan (maxTemperature xs) xs
```

```
Main> hottest yesterday1
["faro"]
Main> hottest yesterday2
["faro","beja"]
Main> hottest (yesterday2 ++ [("funchal",18.5)])
["funchal"]
Main> |
```

A função hottest percorre a lista duas vezes: uma para achar a temperatura máxima e outra para seleccionar as cidades cuja temperatura é maior ou igual à temperatura máxima. O problema pode resolver-se com uma passagem só, mas é menos simples e mais trabalhoso.

Ordenação de listas

- E se quisermos a lista das cidades mais quentes por ordem alfabética?
- Ou, então, se quisermos a lista dos pares cidade – temperatura, por ordem temperatura decendente?
- Temos de ser capazes de ordenar as listas!
- Aprendamos isso, primeiro com listas simples, de cadeias de caracteres.

Como ordenar uma lista de cadeias?

- Escolhemos os elementos estritamente menores do que a cabeça e ordenamos, obtendo uma lista s_1 , ordenada.
- Escolhemos os elementos iguais à cabeça e construímos uma lista s_2 , que está ordenada, uma vez que todos os elementos são iguais.
- Escolhemos os elementos estritamente maiores do que a cabeça e ordenamos, obtendo uma lista s_3 , ordenada.
- Concatenamos s_1 , s_2 e s_3 e já está.

Programando a ordenação

```
smallerThan :: String -> [String] -> [String]
```

```
smallerThan x [] = []
```

```
smallerThan x (y:ys)
```

```
  | y < x = y : z
```

```
  | otherwise = z
```

```
  where z = smallerThan x ys
```

```
largerThan :: String -> [String] -> [String]
```

```
largerThan x [] = []
```

```
largerThan x (y:ys)
```

```
  | y > x = y : z
```

```
  | otherwise = z
```

```
  where z = largerThan x ys
```

```
equalTo :: String -> [String] -> [String]
```

```
equalTo x [] = []
```

```
equalTo x (y:ys)
```

```
  | y == x = y : z
```

```
  | otherwise = z
```

```
  where z = equalTo x ys
```

```
Main> smallerThan "fff" ["rrr", "ddd", "xxx", "aaa", "fff", "rrr",  
"ddd", "bbb", "zzz", "rrr", "xxx", "bbb"]  
["ddd", "aaa", "ddd", "bbb", "bbb"]  
Main> largerThan "fff" ["rrr", "ddd", "xxx", "aaa", "fff", "rrr",  
"ddd", "bbb", "zzz", "rrr", "xxx", "bbb"]  
["rrr", "xxx", "rrr", "zzz", "rrr", "xxx"]  
Main> equalTo "rrr" ["rrr", "ddd", "xxx", "aaa", "fff", "rrr",  
"ddd", "bbb", "zzz", "rrr", "xxx", "bbb"]  
["rrr", "rrr", "rrr"]
```

O quicksort

- Este é o famoso algoritmo *quicksort*, inventado por Hoare, em 1962.

```
qsort :: [String] -> [String]
qsort [] = []
qsort (x : xs) = qsort (smallerThan x xs) ++
                  equalTo x (x : xs) ++
                  qsort (largerThan x xs)
```

```
Main> qsort ["rrr", "ddd", "xxx", "aaa", "fff", "rrr",
"ddd", "bbb", "zzz", "rrr", "xxx", "bbb"]
["aaa", "bbb", "bbb", "ddd", "ddd", "fff", "rrr", "rrr", "rrr", "xxx",
"xxx", "zzz"]
Main> qsort (fst (unzip yesterday1))
["aveiro", "beja", "braganca", "castelo
branco", "evora", "faro", "lisboa", "porto", "setubal", "viseu"]
```

O cálculo do quicksort

```
qsort ["rr","dd","xx","aa","ff","rr","dd","bb","zz","rr","xx","bb"] =  
qsort ["dd","aa","ff","dd","bb","bb"] ++ ["rr","rr","rr"] ++ qsort ["xx","zz","xx"] =  
(qsort ["aa","bb","bb"] ++ ["dd","dd"] ++ qsort ["ff"]) ++ ["rr","rr","rr"] ++  
  qsort ["xx","zz","xx"] =  
((qsort [] ++ ["aa"] ++ qsort["bb","bb"]) ++ ["dd","dd"] ++ qsort ["ff"]) ++ ["rr","rr","rr"]  
  ++ qsort ["xx","zz","xx"] =  
(([] ++ ["aa"] ++ (qsort [] ++ ["bb","bb"] ++ qsort [])) ++ ["dd","dd"] ++ qsort ["ff"]) ++  
  ["rr","rr","rr"] ++ qsort["xx","zz","xx"] =  
(([] ++ ["aa"] ++ ([] ++ ["bb","bb"] ++ [])) ++ ["dd","dd"] ++ qsort ["ff"]) ++ ["rr","rr","rr"]  
  ++ qsort ["xx","zz","xx"] =  
(([] ++ ["aa"] ++ ["bb","bb"]) ++ ["dd","dd"] ++ qsort ["ff"]) ++ ["rr","rr","rr"] ++  
  qsort["xx","zz","xx"] =  
(["aa","bb","bb"]) ++ ["dd","dd"] ++ qsort ["ff"]) ++ ["rr","rr","rr"] ++  
  qsort ["xx","zz","xx"] =  
(["aa","bb","bb"]) ++ ["dd","dd"] ++ (qsort [] ++ ["ff"] ++ qsort [])) ++ ["rr","rr","rr"] ++  
  qsort["xx","zz","xx"] =  
(["aa","bb","bb"]) ++ ["dd","dd"] ++ ([] ++ ["ff"] ++ []) ++ ["rr","rr","rr"] ++  
  qsort ["xx","zz","xx"] =  
(["aa","bb","bb"]) ++ ["dd","dd"] ++ ["ff"]) ++ ["rr","rr","rr"] ++ qsort["xx","zz","xx"] =  
["aa","bb","bb","dd","dd","ff"] ++ ["rr","rr","rr"] ++ qsort ["xx","zz","xx"] =  
...  
["aa","bb","bb","dd","dd","ff","rr","rr","rr","xx","xx","zz"]
```

Exercícios

- Programe uma função para calcular o ponto médio de um segmento definido pelo seus extremos, representando cada ponto por um par de números Double.
- Idem, mas para calcular o par de pontos que dividem o segmento em três partes iguais.
- Adapte a função qsort para ordenar listas de inteiros. Experimente.
- Reprograme a função hottest usando uma só passagem na lista.
- Programe uma função para calcular a amplitude térmica, isto é, a diferença entre a temperatura máxima e a temperatura mínima.
- Programe uma função para calcular a temperatura média.

Controlo

- O tipo dos pares de inteiros é (Int, Int), [Int, Int], ou {Int, Int} ou <Int, Int>?
- Quem inventou o quicksort?
- Que faz a função zip? E a função unzip?
- Qual é a função que dá o primeiro elemento de um par? E o segundo?
- A função find está no prelúdio? E a função qsort?

Na próxima aula

- Estudaremos a ordenação com mais pormenor.
- Em particular, veremos como generalizar, de maneira a que uma única função dê para ordenar listas de vários tipos.

Fundamentos da Programação

Oitava aula:
Funções polimórficas e algoritmos
de ordenação

Nesta aula vamos

- Aprender a programar funções polimórficas, isto é, funções que podem ser chamadas agora com argumentos de um tipo, mais tarde com argumentos de outro.
- A maior parte das funções do prelúdio são polimórficas.

O comprimento

- No prelúdio, a função `length` é polimórfica.
- No entanto, nós programámo-la, com `lengthList`, só para listas de inteiros:

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

```
Main> length [5..12]
8
Main> lengthList [5..12]
8
Main> length ["faro", "tavira", "silves", "portimao",
"lagos"]
5
Main> lengthList ["faro", "tavira", "silves", "portimao",
"lagos"]
ERROR - Type error in application
*** Expression      : lengthList
["faro","tavira","silves","portimao","lagos"]
*** Term           :
["faro","tavira","silves","portimao","lagos"]
*** Type           : [[Char]]
*** Does not match : [Int]
```

O comprimento polimórfico

- Se programássemos uma variante da função `lengthList` para listas de `String`, a definição seria idêntica; apenas a assinatura seria diferente.
- Pois bem, usemos uma assinatura que dê para todos os tipos de listas:

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

```
lengthList [] = 0
```

```
lengthList (_:xs) = 1 + lengthList xs
```

Atenção: aquele *a* é um tipo *polimórfico*. Significa que a definição se aplica qualquer tipo reconhecido.

Experimentando...

```
Main> lengthList [2..6]
5
Main> lengthList ["londres","paris","roma","madrid"]
4
Main> lengthList []
0
Main> lengthList [("aa",3),("zz",22)]
2
Main> lengthList [1.0, 0.5, 0.25, 0.125, 0.0625, 0.03125]
6
Main> lengthList (2,5,6)
ERROR - Type error in application
*** Expression      : lengthList (2,5,6)
*** Term            : (2,5,6)
*** Type            : (b,c,d)
*** Does not match : [a]

Main> lengthList 289800900
ERROR - Cannot infer instance
*** Instance       : Num [a]
*** Expression     : lengthList 289800900
```

Mais exemplos

```
Main> lengthList [1..1000000]
ERROR - C stack overflow
Main> length [1..1000000]
1000000
Main> lengthList [[1..1000000]]
1
Main> lengthList (take 4 [1..1000000])
4
Main> lengthList "vila real de santo antonio"
26
Main> lengthList ["vila real de santo antonio"]
1
Main> |
```

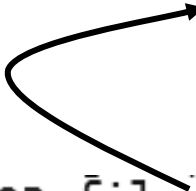
Este demora um pouco, mas consegue!

- Repare, um cadeia, String, é uma lista de caracteres, [Char]:

```
Main> :type "vila real de santo antonio"
"vila real de santo antonio" :: String
Main> :type ["vila real de santo antonio"]
["vila real de santo antonio"] :: [[Char]]
Main> :type ['f','a','r','o']
['f','a','r','o'] :: [Char]
Main> ['f','a','r','o']
"faro"
```

A soma polimórfica

- Apliquemos a mesma técnica à soma, para podermos somar listas seja do que for.



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

```
ERROR file:polymorphic.hs:6 - Inferred type is not general
enough
*** Expression      : sumList
*** Expected type   : [a] -> a
*** Inferred type   : [Integer] -> Integer
```

De facto, da segunda equação inferimos que o tipo `a` tem de ter um elemento zero, o que não é verdade para todos os tipos.

A soma polimórfica, constrangida

- Indicamos que o tipo polimórfico tem de ser uma *instância* de um tipo numérico:

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

```
Main> sumList [1..1000]  
500500  
Main> sumList [1.0, 0.5, 0.25, 0.125, 0.0625, 0.03125]  
1.96875  
Main> sumList ["64", "32", "16"]  
ERROR - Cannot infer instance  
*** Instance      : Num [Char]  
*** Expression   : sumList ["64","32","16"]
```


Classes de tipos

- Num, dos tipos numéricos, com (+), (−), (*).
- Eq, dos tipos com igualdade.
- Ord, dos tipos com ordem.
- Integral, dos tipos numéricos inteiros, isto é, com divisão inteira, div, e resto, mod.
- Fractional, dos tipos numéricos não inteiros, isto é, com divisão “exacta”, (/).

Ord é uma
subclasse de Eq.

A função elemList precisa de tipos
com igualdade e a minimumList
precisa de tipos com ordem.

Integral e Fractional
são subclasses de
Num.

Igualdade...

- A função `elemList` precisa da igualdade. Logo, aplica-se a instâncias da classe `Eq`:

```
elemList :: Eq a => a -> [a] -> Bool  
elemList _ [] = False  
elemList x (y:ys) = x == y || elemList x ys
```

```
Main> elemList 78 [2,4..100]  
True  
Main> elemList 678 [1,3..1000000]  
False  
Main> elemList 679 [1,3..1000000]  
True  
Main> elemList 'r' "tavira"  
True  
Main> elemList [] [[6,7],[9],[],[1..4]]  
True  
Main> elemList ("aa",5) [("zz",2),("rr",9),("aa",5),("ss",3)]  
True  
Main> elemList ("aa",8) [("zz",2),("rr",9),("aa",5),("ss",3)]  
False
```

... e Ordem

- A função `minimumList` usa a função `min`, a qual requer argumentos que sejam instâncias da classe `Ord`:

```
minimumList :: Ord a => [a] -> a
```

```
minimumList [] = error "minimumList: empty list"
```

```
minimumList [x] = x
```

```
minimumList (x:xs) = min x (minimumList xs)
```

```
Main> :type min
```

```
min :: Ord a => a -> a -> a
```

```
Main> minimumList [4,8,3,9,13,4]
```

```
3
```

```
Main> minimumList "silves"
```

```
'e'
```

```
Main> minimumList [[3,7],[7,2,5],[2,10,5],[2,14],[4]]
```

```
[2,10,5]
```

```
Main> minimumList [("zz",2),("rr",9),("hh",5),("ss",3)]
```

```
("hh",5)
```

```
10-0 Main> minimumList ["guarda","viseu","coimbra","leiria"]
```

```
"coimbra"
```

Ordenação geral

- Podemos programar o quicksort para listas de elementos de qualquer tipo, desde que esse tipo seja uma instância da classe Ord.

```
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x : xs) = qsort (smallerThan x xs) ++
                  equalTo x (x : xs) ++
                  qsort (largerThan x xs)
```

Mais pequenos, etc.

- A função dos mais pequenos requer Ord a:

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

- A função dos mais grandes requer Ord a:

```
largerThan :: Ord a => a -> [a] -> [a]
largerThan x [] = []
largerThan x (y:ys)
  | y > x = y : z
  | otherwise = z
  where z = largerThan x ys
```

- A função dos iguais requer Eq a:

```
equalTo :: Eq a => a -> [a] -> [a]
equalTo x [] = []
equalTo x (y:ys)
  | y == x = y : z
  | otherwise = z
  where z = equalTo x ys
```

Estas três funções são semelhantes. Havemos de as unificar.

Experimentando o quicksort genérico

```
Main> qsort [4,7,2,4,5,9,12,9,1,3,8,6]
[1,2,3,4,4,5,6,7,8,9,9,12]
Main> qsort (reverse [1..100])
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25
,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47
,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69
,70,71,72,73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,88,89,90,91
,92,93,94,95,96,97,98,99,100]
Main> qsort "tavira"
"aairtv"
Main> qsort "vila real de santo antonio"
"aaaadeeeiillnnnooorsttv"
Main> qsort [[4,1],[3,2,2],[6],[2,13],[2,5,6,1],[],[2,4,1],[4,9],
[9],[3,(-1)],[6,8]]
[[],[2,4,1],[2,5,6,1],[2,13],[3,-1],[3,2,2],[4,1],[4,9],[6],[6,8],
[9]]
Main> qsort [("ss",6),("ee",3),("tt",8),("ee",6),("hh",1),
("ss",4),("ss",6),("ee",7),("tt",4),("ee",2),("tt",1),("ss",4)]
[("ee",2),("ee",3),("ee",6),("ee",7),("hh",1),("ss",4),("ss",4),
("ss",6),("ss",6),("tt",1),("tt",4),("tt",8)]
Main>
Main> [10,9..1]
[10,9,8,7,6,5,4,3,2,1]
Main> qsort [10,9..1]
[1,2,3,4,5,6,7,8,9,10]
Main> take 3 (qsort [100,99..1])
[1,2,3]
Main> take 3 (qsort [1000,999..1])
```

Mas nem sempre o
quicksort se desembaraça
com êxito ☹.

10-01-2008

ERROR - Garbage collection fails to reclaim sufficient space
Main>

Quicksort descendente

- Basta colocar os mais grandes antes dos iguais, antes dos mais pequenos:

```
qsortDescending :: Ord a => [a] -> [a]
qsortDescending [] = []
qsortDescending (x : xs) =
    qsortDescending (largerThan x xs) ++
    equalTo x (x : xs) ++
    qsortDescending (smallerThan x xs)
```

```
Main> qsortDescending "portimao"
"trpoomia"
Main> take 3 (qsortDescending [1..100])
[100,99,98]
Main> |
```

Ordenação por inserção

- Este é outro algoritmo de ordenação: para ordenar uma lista, ordena-se a cauda e insere-se a cabeça na posição certa da cauda ordenada:

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

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

A função insert insere x antes da primeira ocorrência na lista de um elemento maior ou igual a x. Se a lista estiver ordenada, continua ordenada.

```
Main> insert 13 [2,4..20]
[2,4,6,8,10,12,13,14,16,18,20]
Main> insert 'd' ['a'..'z']
"abcddefghijklmnopqrstuvwxyz"
Main> insert 'p' "lisboa"
"lipsboa"
Main> isort [4,1,8,4,7,9,8,4,4,6]
[1,4,4,4,4,6,7,8,8,9]
Main> isort "liberdade"
"abddeeilr"
Main>
```


Exemplo: anagramas

- Uma função para verificar se duas palavras são anagramas, isto é, se se escrevem com as mesmas letras.

```
anagrams :: Ord a => [a] -> [a] -> Bool
anagrams xs ys = qsort xs == qsort ys
```

```
Main> anagrams "rato" "rota"
True
Main> anagrams "lobo" "bola"
False
Main> anagrams "almirante" "alimentar"
True
Main> anagrams "corpo" "porco"
True
Main> anagrams [1,3..9] [9,7..1]
True
Main> anagrams "faro" "faro]"
False
Main> anagrams [[3,5],[6],[],[4,44]] [[],[6],[4,44],[3,5]]
True
Main> anagrams [[3,5],[6],[],[4,44]] [[],[6],[44,4],[3,5]]
False
Main> |
```

O problema é formulado em termos de cadeias de caracteres mas, como generalizámos para instâncias de Ord, pudemos logo usar para listas de inteiros e para listas de listas.

Exercícios

- Reveja as funções das listas de exercícios e generalize-as, sempre que possível.

Controlo

- Que classes de tipos já conhecemos?
- Na função para calcular a média aritmética de uma lista, os elementos da lista devem ser de um tipo instância de que classe?
- Qual é o algoritmo de ordenação mais utilizado: quicksort ou insertion sort?
- Qual é a lista mínima, isto é, aquela que é menor que todas as outras? E a lista máxima? ;-)
- Qual das duas é menor [1..100000] e [2]?
- O que são anagramas?

Na próxima aula...

- Estudaremos outros casos de herança, sem figuras.
- Usaremos exemplos com pessoas e amigos.

Fundamentos da Programação

Nona aula:

Ordenação de listas de pares

Nesta aula vamos...

- Retomar o problema das temperaturas e aprender a ordenar listas de pares, de várias maneiras.
- Encontrar as listas em compreensão.
- Resolver, como exercício, alguns outros problemas.

Recordando

- Temos as temperaturas registadas numa listas de pares.
- Já calculámos a temperatura máxima, a lista das temperaturas de uma dada cidade e a lista das cidades mais quentes.
- Queremos agora listar os pares por ordem decrescente de temperatura.

Para aquecer: generalizar o find

- Antes programámos para uma lista de pares [(String, Double)].
- Agora programamos para uma lista de pares [(a,b)] onde a é uma instância da classe Eq:

```
find :: Eq a => a -> [(a, b)] -> [b]
find _ [] = []
find x ((a,b):ts)
  | x == a = b : z
  | otherwise = z
  where
    z = find x ts
```


Ordenando os pares

- Se ordenarmos os pares cidade-temperatura directamente, a lista fica por ordem alfabética das cidades:

```
Main> yesterday2
[("lisboa",15.3),("setubal",17.7),("evora",14.8),("castelo
branco",12.2),("faro",18.2),("viseu",17.7),("porto",11.7),
("aveiro",14.2),("beja",18.2),("braganca",9.6)]
Main> qsort yesterday2
[("aveiro",14.2),("beja",18.2),("braganca",9.6),("castelo
branco",12.2),("evora",14.8),("faro",18.2),("lisboa",15.3),
("porto",11.7),("setubal",17.7),("viseu",17.7)]
Main> s2
[("ggg",10.0),("sss",11.0),("ttt",9.0),("ggg",14.0),("fff",13.0),
("ttt",10.0),("hhh",9.0),("jjj",12.0),("hhh",11.0),("sss",10.0),
("kkk",12.0),("zzz",11.0),("jjj",10.0),("aaa",10.0),("ggg",11.0),
("sss",12.0),("zzz",13.0),("xxx",14.0),("bbb",14.0),("ggg",10.0),
("sss",9.0)]
Main> qsort s2
[("aaa",10.0),("bbb",14.0),("fff",13.0),("ggg",10.0),("ggg",10.0),
("ggg",11.0),("ggg",14.0),("hhh",9.0),("hhh",11.0),("jjj",10.0),
("jjj",12.0),("kkk",12.0),("sss",9.0),("sss",10.0),("sss",11.0),
("sss",12.0),("ttt",9.0),("ttt",10.0),("xxx",14.0),("zzz",11.0),
("zzz",13.0)]
```

Ideia



- Trocamos os elementos dentro dos pares, ordenamos decendentemente e trocamos de novo.
- Para trocar, programamos genericamente:

```
swapPairs :: [(a,b)] -> [(b,a)]  
swapPairs [] = []  
swapPairs ((x,y):zs) = (y,x) : swapPairs zs
```

- Agora é simples:

```
temperatureDown :: [(String,Double)] -> [(String,Double)]  
temperatureDown xs =  
    swapPairs (qsortDescending (swapPairs xs))
```

Função composta

- Observe:

```
temperatureDown :: [(String,Double)] -> [(String,Double)]  
temperatureDown xs =  
    swapPairs (qsortDescending (swapPairs xs))
```

- Isto quer dizer que a função temperatureDown igual à função composta swapPairs ◦ qsortDescending ◦ swapPairs. Por isso a definição pode ficar assim, usando o operador ponto, de função composta:

```
temperatureDown :: [(String,Double)] -> [(String,Double)]  
temperatureDown = swapPairs . qsortDescending . swapPairs
```

Mas há um pequeno problema...

- Cidades com a mesma temperatura vêm por ordem descendente também, mas não é isso que convém ☹:

```
Main> temperatureDown yesterday2  
[("faro",18.2),("beja",18.2),("viseu",17.7),("setubal",17.7),  
("lisboa",15.3),("evora",14.8),("aveiro",14.2),("castelo  
branco",12.2),("porto",11.7),("braganca",9.6)]  
Main> temperatureDown s2  
[("xxx",14.0),("ggg",14.0),("bbb",14.0),("zzz",13.0),("fff",13.0),  
("sss",12.0),("kkk",12.0),("jjj",12.0),("zzz",11.0),("sss",11.0),  
("hhh",11.0),("ggg",11.0),("ttt",10.0),("sss",10.0),("jjj",10.0),  
("ggg",10.0),("ggg",10.0),("aaa",10.0),("ttt",9.0),("sss",9.0),  
("hhh",9.0)]
```

- Isto é, as cidades com a mesma temperatura deviam ter sido ordenadas normalmente.

Adaptamos o quicksort

- Ordenamos por temperatura descendente os pares mais quentes do que a cabeça; concatenamos os pares com a mesma temperatura que a cabeça, ordenados normalmente; concatenamos os pares mais frios do que a cabeça, ordenados decendentemente:

```
qsortTempDown :: [(String,Double)] -> [(String,Double)]
qsortTempDown [] = []
qsortTempDown (x : xs) =
    qsortTempDown (warmerThan x xs) ++
    qsort (sameTemperature x (x:xs)) ++
    qsortTempDown (colderThan x xs)
```

Confirmamos...

```
Main> qsortTempDown s2
[("bbb",14.0),("ggg",14.0),("xxx",14.0),("fff",13.0),("zzz",13.0),
("jjj",12.0),("kkk",12.0),("sss",12.0),("ggg",11.0),("hhh",11.0),
("sss",11.0),("zzz",11.0),("aaa",10.0),("ggg",10.0),("ggg",10.0),
("jjj",10.0),("sss",10.0),("ttt",10.0),("hhh",9.0),("sss",9.0),
("ttt",9.0)]
Main> qsortTempDown yesterday2
[("beja",18.2),("faro",18.2),("setubal",17.7),("viseu",17.7),
("lisboa",15.3),("evora",14.8),("aveiro",14.2),("castelo
branco",12.2),("porto",11.7),("braganca",9.6)]
Main> qsortTempDown (yesterday2 ++ s2)
[("beja",18.2),("faro",18.2),("setubal",17.7),("viseu",17.7),
("lisboa",15.3),("evora",14.8),("aveiro",14.2),("bbb",14.0),
("ggg",14.0),("xxx",14.0),("fff",13.0),("zzz",13.0),("castelo
branco",12.2),("jjj",12.0),("kkk",12.0),("sss",12.0),
("porto",11.7),("ggg",11.0),("hhh",11.0),("sss",11.0),
("zzz",11.0),("aaa",10.0),("ggg",10.0),("ggg",10.0),("jjj",10.0),
("sss",10.0),("ttt",10.0),("braganca",9.6),("hhh",9.0),
("sss",9.0),("ttt",9.0)]
Main>
```

Funções warmer, etc.

- São funções de filtragem, que seleccionam de uma lista os elementos que verificam uma certa propriedade: serem mais quentes do que um elemento dado, etc.

```
warmerThan :: (String, Double) -> [(String,Double)] -> [(String,Double)]
warmerThan x [] = []
warmerThan x (y:ys)
  | snd y > snd x = y : z
  | otherwise = z
  where z = warmerThan x ys
```

As funções colderThan e sameTemperature são parecidas.

```
Main> warmerThan ("aa",15.0) yesterday2
[("lisboa",15.3),("setubal",17.7),("faro",18.2),("viseu",17.7),
 ("beja",18.2)]
Main> warmerThan ("aa",20.0) yesterday2
[]
```

Listas em compreensão

- Podemos em Haskell construir uma lista a partir de outra, usando o mecanismo das listas em compreensão.
- Dada uma lista `xs`, exprimimos directamente a ideia “a lista dos elementos de `xs` tais que ...”
- Observe, para a função `warmerThan`:

```
warmerThan ::
```

```
(String, Double) -> [(String,Double)] -> [(String,Double)]
```

```
warmerThan x ys = [y | y <- ys, snd y > snd x]
```


Mais exemplos de compreensão

- A lista dos x primeiros quadrados perfeitos:

```
squares :: Int -> [Int]
squares n = [x*x | x <- [1..n]]
```

- A lista dos divisores próprios de x :

```
divisors :: Int -> [Int]
divisors x = [x' | x' <- [1..div x 2], mod x x' == 0]
```

```
Main> squares 7
[1,4,9,16,25,36,49]
Main> divisors 48
[1,2,3,4,6,8,12,16,24]
Main> |
```

É bem mais simples
do que de outro modo!

Antes, faríamos assim

- Os quadrados perfeitos:

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

```
squares' :: Int -> [Int]
squares' x = squaresFrom [1..x]
```

- A lista dos divisores próprios de x:

```
divisorsFrom :: Int -> [Int] -> [Int]
divisorsFrom _ [] = []
divisorsFrom x (y:ys)
    | mod x y == 0 = y : t
    | otherwise = t
```

where

t = divisorsFrom x ys

```
Main> squares' 12
[1,4,9,16,25,36,49,64,81,100,121,144]
Main> divisors' 500
[1,2,4,5,10,20,25,50,100,125,250]
Main> |
```

```
divisors' :: Int -> [Int]
divisors' x = divisorsFrom x [2..div x 2]
```

Simplificando a programação

- A técnica das listas em compreensão simplifica bastante a programação de muitos casos comuns.
- De que outras simplificações precisamos ainda?
- Precisamos de poder dar um nome aos tipos de dados estruturados, para não ter de escrever sempre [(String, Double)], por exemplo.
- Precisamos de uma maneira de seleccionar elementos de uma lista, parametrizada (essa maneira) pelo critério de selecção.
- Precisamos de programar um quicksort genérico (isto é, que dê para vários tipos) e geral, isto é, que aceite parametricamente a relação de ordem que queremos estabelecer na lista.

Problema da nota dos questionários

- Dada a lista das notas nos questionários e o número de questionários.

```
quizPoints :: [Double] -> [Int]
quizPoints xs = [if x >= 7.0 then 1 else 0 | x <- xs]

divExact :: Int -> Int -> Double
divExact x y = fromIntegral x / fromIntegral y

quizGradeExact :: [Double] -> Int -> Double
quizGradeExact xs y = divExact (sum (quizPoints xs)) y

roundToFraction :: Double -> Int -> Double
roundToFraction x y = divExact (round (x * fromIntegral y)) y

quizGrade :: [Double] -> Int -> Double
quizGrade xs y = roundToFraction (quizGradeExact xs y) 5
```

Problema da nota final

- Dadas a notas das cadeiras e os pesos, na forma de uma listas de pares, calcular a média ponderada arredondada.

```
finalGrade :: [(Double, Double)] -> Int
```

```
finalGrade = round . finalGradeExact
```

```
finalGradeExact :: [(Double, Double)] -> Double
```

```
finalGradeExact zs = sum [x*y | (x,y) <- zs] / sum (snd (unzip zs))
```

Exercícios

- Programe uma função para contar as cidades com temperatura superior à média.
- Programe uma função para produzir um histograma das temperaturas arredondadas. O resultado é uma lista de pares [(Int, Int)] onde cada par (x,y) significa que houve y cidades com temperatura x. Só queremos pares com $y > 0$.
- Programe uma função para calcular uma lista de números primos, usando listas em compreensão.
- Programe uma função para calcular a lista das potências de expoente inteiro de um dado número até um dado expoente.

Controlo

- Como é a assinatura do `find` polimórfico?
- Quais são as particularidades da ordenação de pares?
- Qual é o operador de composição de funções em Haskell?
- Qual é o aspecto sintáctico das listas em compreensão?
- Como é que se arredonda a uma fracção da unidade? E a um múltiplo da unidade?

Na próxima aula...

- Aprenderemos a guardar os resultados dos nossos programas em ficheiros.
- E logo a seguir, aprenderemos a ler os argumentos dos nossos programas a partir de ficheiros.

Fundamentos da Programação

Décima aula:
Lendo e escrevendo

Nesta aula vamos

- Aprender a escrever na consola.
- Logo a seguir, aprender a ler da consola.
- De facto, até agora os nossos programas não tinham input ou output.
- Temos usado o Hugs como uma calculadora funcional: chamamos uma função, passando os argumentos, e ele mostra o resultado, automaticamente.

Escrevendo e Lendo

- Frequentemente, queremos guardar os resultados do nosso programa, para podermos consultá-los mais tarde, ou para servirem de dados em outros programas.
- Também temos de ser capazes de utilizar nos nossos programa dados guardados em ficheiros por outros programas (programas em Haskell ou noutras linguagens) ou inseridos por alguém usando um editor de texto.

A consola

- Para já, vamos aprender a escrever na “consola” e a ler da consola.
- A seguir, vamos conseguir escrever em ficheiros e ler de ficheiros simplesmente redirigindo a saída-padrão e a entrada-padrão
- Finalmente, aprenderemos a especificar directamente os ficheiros que queremos usar.

Hello World!

- O primeiro programa, em qualquer linguagem é o “Hello World”: apenas escreve na consola a mensagem “Hello World!”.
- Não se trata apenas de fazer aparecer “Hello World!”, o que seria trivial, mas de escrever um programa faz aparecer na consola a mensagem “Hello World!”, sem as aspas.

```
Main> "Hello world!"  
"Hello world!"  
Main>
```

Hello World, em Haskell

- Primeiro, uma função para escrever HelloWorld na consola. Observe:

```
helloWorld :: IO ()  
helloWorld = putStrLn "Hello World!"
```

- Chamamo-la no WinHugs e ela escreve, sem aspas, mudando de linha no final:

```
Main> helloWorld  
Hello world!
```

Também há uma função putStr, que escreve e fica na mesma linha.

```
Main> |
```

A função main

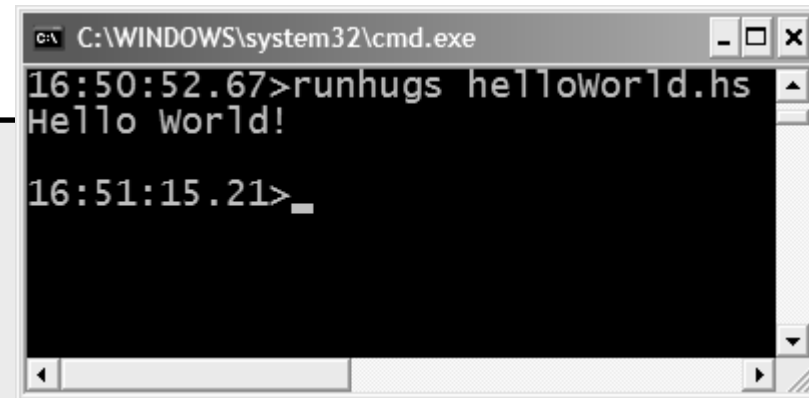
- A função main, de tipo IO (), é chamada quando corremos um programa Haskell na linha de comando, através do runhugs:

```
main :: IO ()
```

```
main = helloWorld
```

```
helloWorld :: IO ()
```

```
helloWorld = putStrLn ("Hello World!")
```



The screenshot shows a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The prompt is at "16:50:52.67>". The user has entered "runhugs helloworld.hs", and the output "Hello world!" is displayed on the next line. The prompt is now at "16:51:15.21>".

Simplificando, para exemplificar

- Para efeitos do HelloWorld, podíamos dispensar a função helloWorld e fazer tudo na função main:

```
main :: IO()  
main = putStrLn ("Hello World!")
```

- No entanto, em geral, como regra de estilo, programaremos a função main apenas chamando uma de várias funções IO () que haverá no programa, estas sim, fazendo o trabalho interessante.

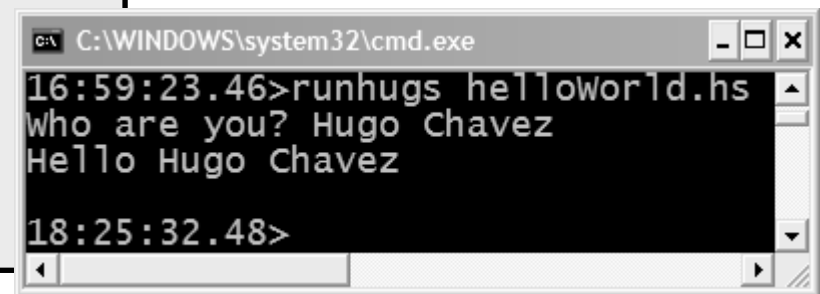
Hello, You!

- Agora queremos um “hello” personalizado.
- A função pede o nome do utilizador e depois ecoa “Hello” seguido do nome.
- Observe:

```
helloYou :: IO ()  
helloYou =  
  do  
    putStr "Who are you? "  
    name <- getLine  
    putStrLn ("Hello " ++ name)
```

```
Main> helloYou  
Who are you? Jack the Ripper  
Hello Jack the Ripper
```

```
Main>
```



The screenshot shows a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The prompt is "16:59:23.46>runhugs helloWorld.hs". The output of the program is "who are you? Hugo Chavez" followed by "Hello Hugo Chavez". The prompt then changes to "18:25:32.48>".

do

- Atenção, a notação `do` só se usa em funções de tipo `IO ()`.
- Serve para definir uma sequência de comandos.
- Todas esses comandos devem estar alinhadas, uma em cada linha.
- Os comandos são funções.
- Um comando que retorna um objecto de tipo `T` tem tipo `IO T`.
- `IO ()` é o tipo dos comandos que não retornam nada.
- Nem pense em usar o `do` nas suas funções “normais”.
- Mais exemplos a seguir.

<-

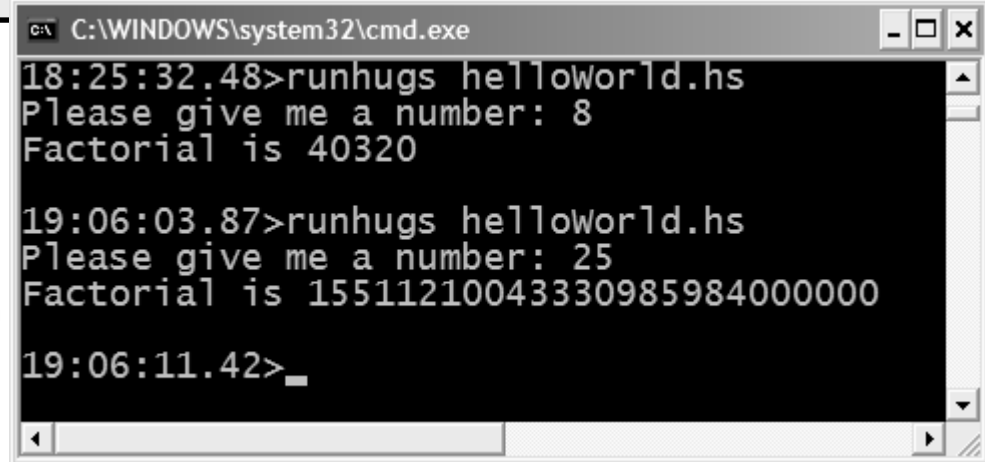
- Note o operador <- em `name <- getLine`.
- É <-, não é =.
- Tecnicamente, a função `getLine` é um comando de tipo `IO String`.
- A função `getLine`, faz uma operação de leitura e retorna uma cadeia, a qual vem “embrulhada” no resultado, que é um objecto de tipo `IO String`.
- Em geral, a sintaxe `x <- y`, para `y` de tipo `IO T`, extrai do resultado de `y` o valor de tipo `T` que lá vem “dentro”, para ser usada no resto da sequência, através no nome `x`.

Hello, Factorial

- E agora, um exemplo com leitura de números.
- A função lê um número e mostra o seu factorial.

```
helloFactorial :: IO ()  
helloFactorial =  
  do  
    putStr "Please give me a number: "  
    s <- getLine  
    putStrLn ("Factorial is " ++ show (factorial (read s)))
```

```
Main> helloFactorial  
Please give me a number: 4  
Factorial is 24  
  
Main> helloFactorial  
Please give me a number: 20  
Factorial is 2432902008176640000  
  
Main>
```



The screenshot shows a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". It displays two successful runs of a Haskell program named "runhugs helloworld.hs". In the first run, the user enters "8" and the output is "Factorial is 40320". In the second run, the user enters "25" and the output is "Factorial is 15511210043330985984000000". The prompt is currently at "19:06:11.42>".

```
C:\WINDOWS\system32\cmd.exe  
18:25:32.48>runhugs helloworld.hs  
Please give me a number: 8  
Factorial is 40320  
  
19:06:03.87>runhugs helloworld.hs  
Please give me a number: 25  
Factorial is 15511210043330985984000000  
  
19:06:11.42>
```

show, read

- A função `show` converte valores para uma representação canónica do tipo `String`.
- A função `read` converte valores de tipo `String` para valores de outro tipo, consoante o contexto.

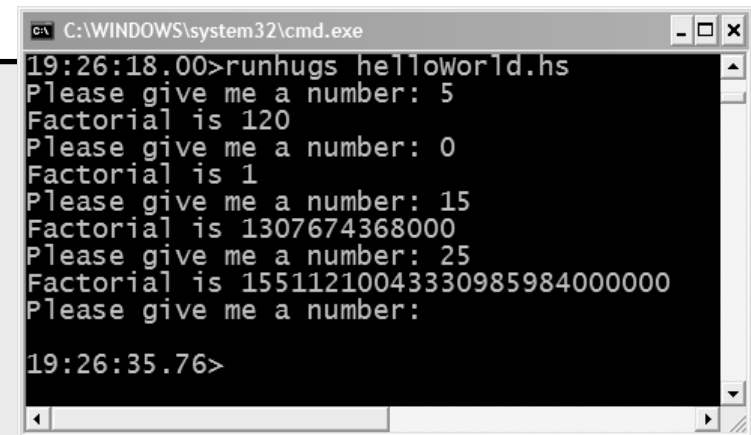
```
Main> show (factorial 10)
"3628800"
Main> factorial (read "10")
3628800
Main> show (factorial (read "10"))
"3628800"
Main>
```

Note bem: estas são funções “normais”, que podemos usar livremente nas nossas funções “normais”.

Hello, Factorial Many

- O função helloFactorial só dá um factorial de cada vez.
- Observe a função helloFactorialMany, que calcula ciclicamente, terminando com um linha vazia:

```
helloFactorialMany :: IO ()
helloFactorialMany =
  do
    putStr "Please give me a number: "
    s <- getLine
    if s /= "" then
      do
        putStrLn ("Factorial is " ++ show (factorial (read s)))
        helloFactorialMany
    else
      return ()
```



```
C:\WINDOWS\system32\cmd.exe
19:26:18.00>runhugs helloworld.hs
Please give me a number: 5
Factorial is 120
Please give me a number: 0
Factorial is 1
Please give me a number: 15
Factorial is 1307674368000
Please give me a number: 25
Factorial is 15511210043330985984000000
Please give me a number:
19:26:35.76>
```

A função main

- A função main chama uma das outras.
- As restantes ficam em comentário:

```
main :: IO()
-- main = helloWorld
-- main = helloYou
-- main = helloFactorial
main = helloFactorialMany
```

Assim podemos fazer as sucessivas experiências, sem estragar as anteriores.

Controlo

- Qual foi o tema da aula de hoje?
- Qual é o tipo da função main?
- Qual a função para ler uma linha inteira da consola?
- Quais são as funções de conversão de e para String?
- Qual a palavra chave que encabeça uma sequência de comandos?
- Como se faz para ler ciclicamente?

Exercícios

- Programe uma função que pergunte ao utilizador o nome e a hora e responda “Bom dia”, “Boa tarde” ou “Boa noite”, consoante a hora seja entre as 4:00 e 11:59, 12:00 e 19:59, e 20:00 e 3:59, respectivamente.
- Modifique a função do helloFactorialMany de maneira a que a partir da segunda vez a mensagem seja “Please give me another number:”
- Programe uma função que pergunte ao utilizador o nome; depois se for um nome de uma dada lista de nomes femininos responde “Olá, minha querida!”; se for de uma lista de nomes masculinos responde “Olá, pá!”; se não, nem uma coisa nem outra, responde “Olá!” só.
- Programe uma função que aceite uma sequência de números inteiros, um em cada linha, terminada por uma linha em branco e que mostre a soma, depois de os ter guardado numa lista.

Na próxima aula

- Vamos aprender a ler os dados não apenas da consola mas de ficheiros em geral.
- E vamos também aprender a escrever os resultados em ficheiros, e não apenas na consola.

Fundamentos da Programação

Décima segunda aula:
Entradas e saídas, de novo

Nesta aula vamos...

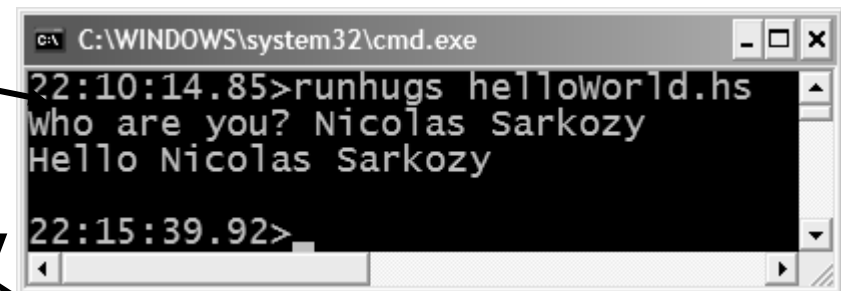
- Estudar mais algumas técnicas para realizar a entrada e saída de dados.
- De passagem, ver outras questões interessantes sobre programação funcional.

Hello quê?

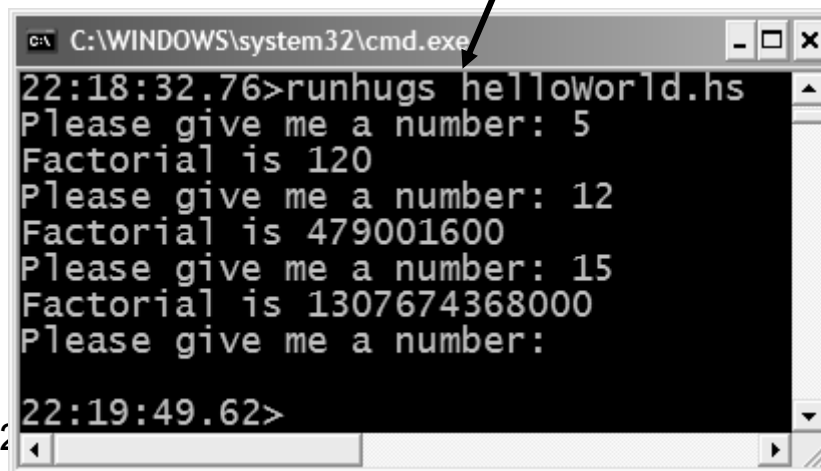
- Hello World.
- Hello You.
- Hello Factorial.
- Hello Factorial Many



```
C:\WINDOWS\system32\cmd.exe
22:16:36.84>runhugs helloworld.hs
Hello World!
22:16:41.12>
```



```
C:\WINDOWS\system32\cmd.exe
22:10:14.85>runhugs helloworld.hs
who are you? Nicolas Sarkozy
Hello Nicolas Sarkozy
22:15:39.92>
```



```
C:\WINDOWS\system32\cmd.exe
22:18:32.76>runhugs helloworld.hs
Please give me a number: 5
Factorial is 120
Please give me a number: 12
Factorial is 479001600
Please give me a number: 15
Factorial is 1307674368000
Please give me a number:
22:19:49.62>
```



```
C:\WINDOWS\system32\cmd.exe
22:16:41.12>runhugs helloworld.hs
Please give me a number: 20
Factorial is 2432902008176640000
22:18:32.76>
```

Hello List

- Problema: ler uma sequência de números, como no Hello Factorial Many, e calcular a sua soma no fim.
- A diferença é que temos de guardar os sucessivos números.
- Usaremos uma lista para isso, é claro.

Função helloList

```
getNumbers :: IO [Int]
getNumbers = ...

helloList :: IO ()
helloList =
    do
        xs <- getNumbers
        print xs
        print (sum xs)
```

Escrevemos os
números, só
para controlar...

Hello Int

- Exercício: ler um int de uma linha, onde não há mais nada e escrever a lista de todos os números de 1 até ao número lido:

```
getInt :: IO Int
getInt =
  do
    s <- getLine
    return (read s)
```

```
helloInt :: IO()
helloInt =
  do
    x <- getInt
    print [1..x]
```

```
Main> helloInt
6
[1,2,3,4,5,6]
```

```
Main> helloInt
4
[1,2,3,4]
```

```
Main> helloInt
8 2
```

Erro, há mais coisas na linha.

```
Program error: Prelude.read: no parse
```

```
Main>
```


Função getNumbers

- Seguimos a estrutura do helloFactorialMany:

```
getNumbers :: IO [Int]
getNumbers =
  do
    putStr "Please give me a number: "
    s <- getLine
    if s /= "" then
      do
        xs <- getNumbers
        return (read s : xs)
    else
      return []
```

Testando o
helloList.

```
Please give me a number: 5
Please give me a number: 2
Please give me a number: 44
Please give me a number: 12
Please give me a number:
[5,2,44,12]
63
Main>
```

Função getNumbersAlt

- Técnica alternativa: transportar a lista em argumento:

```
getNumbersAlt :: [Int] -> IO [Int]
```

```
getNumbersAlt xs =
```

```
  do
```

```
    putStr "Please give me a number: "
```

```
    s <- getLine
```

```
    if s /= "" then
```

```
      getNumbersAlt (read s : xs)
```

```
    else
```

```
      return xs
```

```
helloListAlt :: IO ()
```

```
helloListAlt =
```

```
  do
```

```
    xs <- getNumbersAlt []
```

```
    print xs
```

```
    print (sum xs)
```

```
Main> helloListAlt
```

```
Please give me a number: 7
```

```
Please give me a number: 2
```

```
Please give me a number: 99
```

```
Please give me a number:
```

```
[99,2,7]
```

```
108
```

```
Main>
```

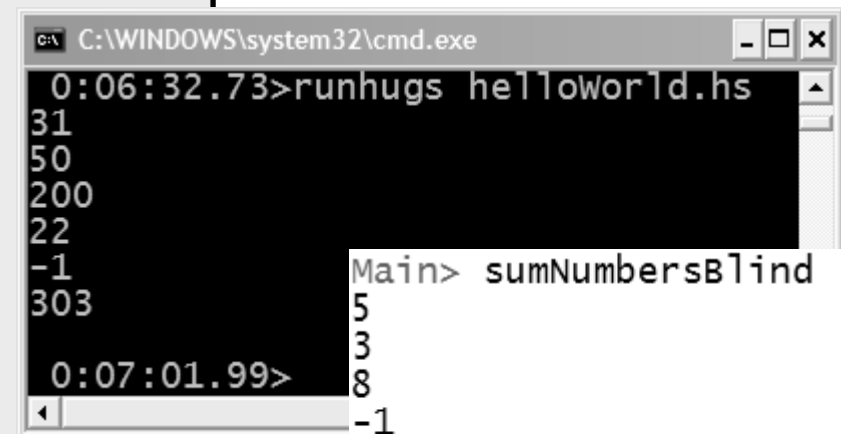
Ah, os números
vêm por ordem
inversa da da
leitura.

Leitura cega

- Problema: ler uma sequência de números terminados por -1 e calcular a soma.

```
getNumbersBlind :: IO [Int]
getNumbersBlind =
  do
    x <- getInt
    if x /= -1 then
      do
        xs <- getNumbersBlind
        return (x : xs)
    else
      return []
```

```
sumNumbersBlind :: IO ()
sumNumbersBlind =
  do
    xs <- getNumbersBlind
    print (sum xs)
```

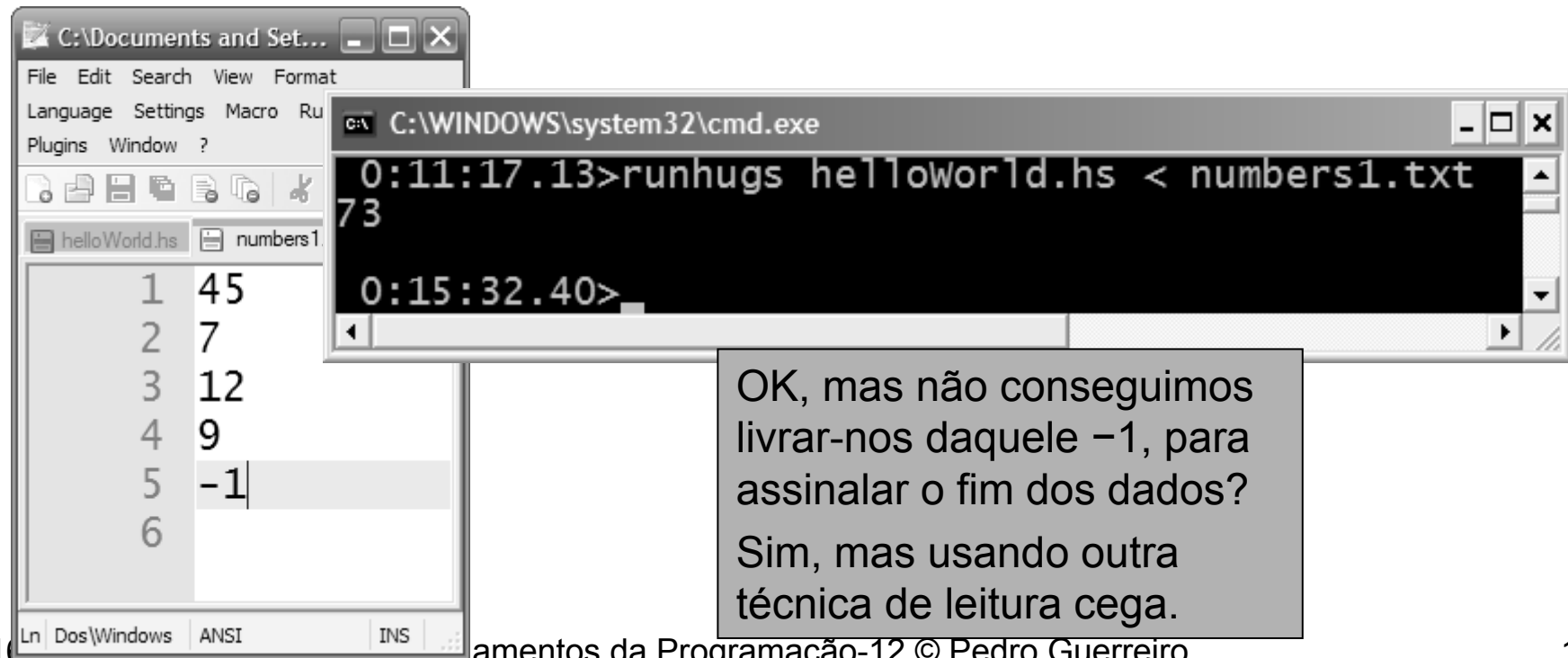


```
C:\WINDOWS\system32\cmd.exe
0:06:32.73>runhugs helloWorld.hs
31
50
200
22
-1
303
0:07:01.99>
```

```
Main> sumNumbersBlind
5
3
8
-1
16
Main>
```

Redirigindo o input

- Na consola do Windows, podemos redirigir o input, de maneira a que a leitura seja feita a partir do ficheiro indicado.

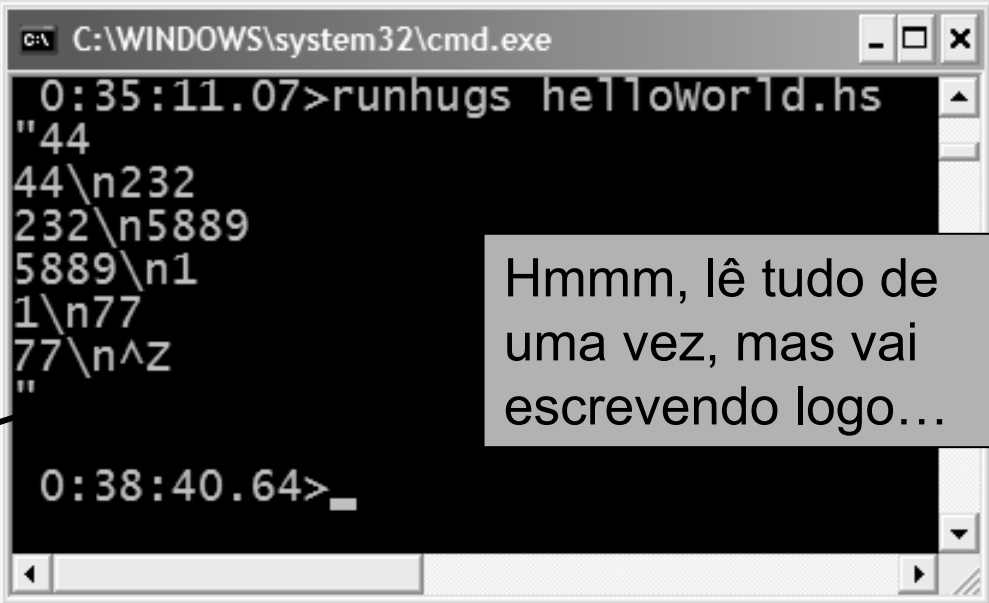


getContents

- A função `getContents` lê o input todo, “de uma vez”, para uma `String`.

```
contents1 :: IO ()
contents1 =
  do
    s <- getContents
    print s
```

Note bem: `^Z` significa que teclamos `Ctrl+Z`, para assinalar o fim dos dados.



```
C:\WINDOWS\system32\cmd.exe
0:35:11.07>runhugs helloworld.hs
"44
44\\n232
232\\n5889
5889\\n1
1\\n77
77\\n^Z
"
0:38:40.64>
```

Hmmm, lê tudo de uma vez, mas vai escrevendo logo...

Redirigindo o output

- Separemos a escrita da leitura, redirigindo o output para um ficheiro

The screenshot shows a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe" and a text editor window titled "C:\Documents and Settings\Pedro Guerreiro\My Documents\My Progra...".

In the command prompt, the following commands are entered:

```
0:38:40.64>runhugs helloWorld.hs > out.txt
```

The output of the program is redirected to the file "out.txt". The output is displayed in the command prompt as:

```
33
56
223
-1
44
^Z
```

Then, the command prompt shows the command:

```
0:43:21.20>type out.txt
```

The output of the "type" command is displayed as:

```
"33\n56\n223\n-1\n44\n"
```

The text editor window shows the contents of the file "out.txt". The text is:

```
1 "33\n56\n223\n-1\n44\n"
2
```

An arrow points from the text in the text editor to a callout box that says:

Isto mostra bem que o resultado do `getContents` é uma String.

Função lines

- Problema: somar todos os números de um ficheiro, onde vêm um por linha.
- Temos de transformar a String lida com `getContents` numa lista de números.
- Primeiro, partimos a String numa lista de Strings, uma para cada linha, com a função `lines`.

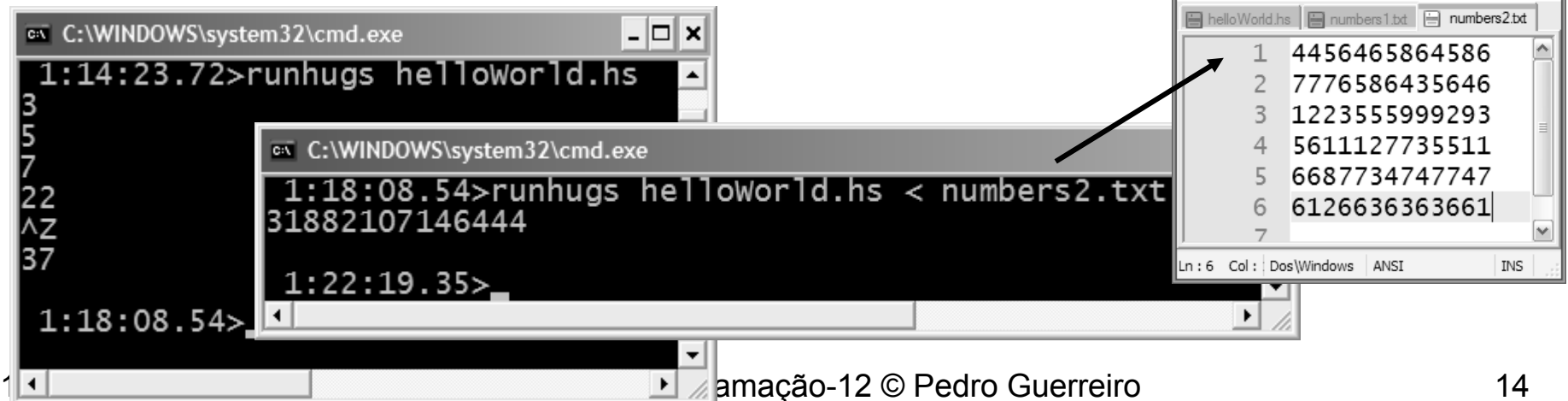
- Observe:

```
Main> length "33\n56\n223\n-1\n44\n"
16
Main> lines "33\n56\n223\n-1\n44\n"
["33","56","223","-1","44"]
Main> lines "lisboa\nporto\nfaro\n"
["lisboa","porto","faro"]
Main>
```

Função map

- A seguir, transformamos cada cadeira numérica num número, usando a função map, e somamos a lista:

```
sumInput :: IO ()
sumInput =
  do
    s <- getContents
    print (sum (map read (lines s)))
```



Função map, definição

- A função map aplica uma dada função a todos os elementos de uma lista:

```
mapList :: (a -> b) -> [a] -> [b]
mapList f xs = [f x | x <- xs]
```

```
Main> map (+1) [10..13]
[11,12,13,14]
Main> map (>5) [3,5,7,9]
[False,False,True,True]
Main> map length [[2,4],[],[10..33]]
[2,0,24]
Main> map (`mod` 2) [5,13,4,1,8]
[1,1,0,1,0]
Main> map (*2) [8,1,9]
[16,2,18]
Main>
```

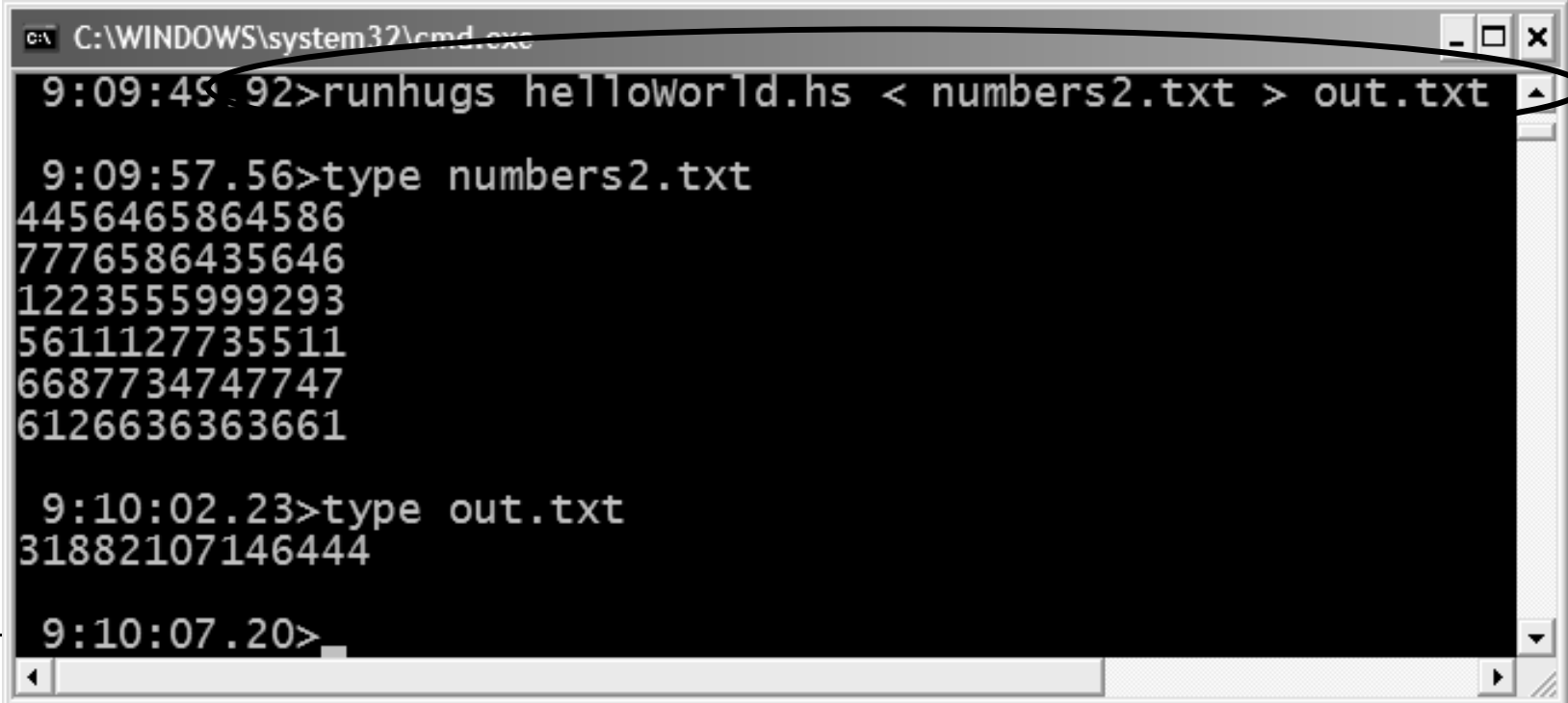
Secções

- Uma secção é uma operação binária à qual falta dum dos operandos.
- Representa uma função unária, em que o operando em falta é o argumento da função, por assim dizer.

```
Hugs> (*10) 21
210
Hugs> (1/) 5
0.2
Hugs> (/2) 15
7.5
Hugs> (`mod` 10) 2008
8
Hugs> ("mini " ++) "cerveja"
"mini cerveja"
Hugs> (*3) 1000
3000
Hugs> (`max` 0) 12
12
Hugs> (`max` 0) (-3)
0
Hugs>
```

Redirigindo o input e o output

- Se rediregirmos o input e o output na linha de comandos, parece que nada acontece, mas os dados são lidos do ficheiro de entrada e os resultados escritos no de saída.



The screenshot shows a Windows command prompt window with the title bar "C:\WINDOWS\system32\cmd.exe". The command prompt displays the following sequence of commands and output:

```
9:09:49.92>runhugs helloWorld.hs < numbers2.txt > out.txt

9:09:57.56>type numbers2.txt
4456465864586
7776586435646
1223555999293
5611127735511
6687734747747
6126636363661

9:10:02.23>type out.txt
31882107146444

9:10:07.20>
```

A black oval is drawn around the command `runhugs helloWorld.hs < numbers2.txt > out.txt` in the first line of the command prompt.

Controlo

- Como se chama função que aplica uma função dada a todos os elementos de uma lista?
- Como se redirige o input na consola? E o output?
- O que são secções?
- Para que serve a função map?
- O que é que a função lines tem a ver com linhas?
- Se houvesse uma função unlines, inversa de lines, que faria ela?

Exercícios

- Escreva uma função contar o número de linhas de um ficheiro.
- E outra para contar o número de caracteres.
- Programe a função mapList recursivamente.

Na próxima aula

- Veremos mais técnicas de leitura e escrita.

Fundamentos da Programação

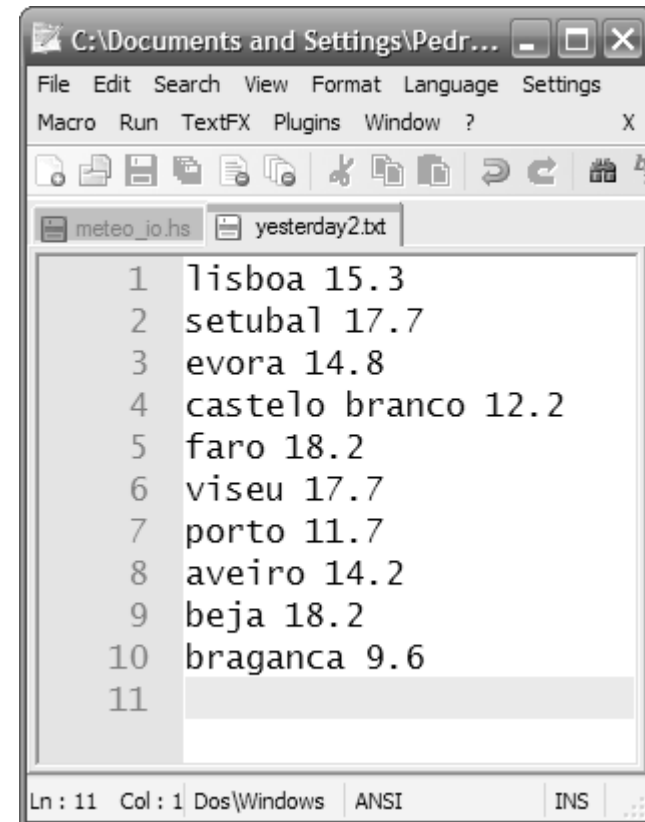
Décima terceira aula:
Usando ficheiros nos programas

Nesta aula vamos...

- Regressar ao problema das temperaturas, agora com os dados lidos a partir de ficheiros.
- Dar uns toques nos programas anteriores, para os tornar mais simples, tirando partido das novidades mais recentes.

Primeira questão, ler os dados

- Temos as observações num ficheiro.
- Queremos criar a lista das observações, que é uma lista de pares.



A screenshot of a text editor window titled 'C:\Documents and Settings\Pedr...'. The window displays a file named 'yesterday2.txt' with the following content:

```
1 lisboa 15.3
2 setubal 17.7
3 evora 14.8
4 castelo branco 12.2
5 faro 18.2
6 viseu 17.7
7 porto 11.7
8 aveiro 14.2
9 beja 18.2
10 braganca 9.6
11
```

The status bar at the bottom indicates 'Ln : 11 Col : 1 Dos\Windows ANSI INS'.

Novos tipos

- Usemos nomes para os tipos apropriados para o domínio do problema:

```
type City = String
type Cities = [City]
type Temperature = Double
type Observation = (City, Temperature)
type Observations = [Observation]
```

- Estes nomes tornam o programa mais simples de entender. Por exemplo:

```
hottest :: [(String,Double)] -> [String]
hottest xs = ...
```

Antes

```
hottest :: Observations -> Cities
hottest xs = ...
```

Agora

Ler as temperaturas

- Lemos com `getContents`.
- Depois, com a função `lines`, separamos as linhas.
- Precisamos de uma função para transformar uma cadeia, por exemplo “tavira 12.5”, no par (“tavira”,12.5).
- Sem esquecer que o nome de certas cidades tem mais do que uma palavra: “vila real 17.2” deve dar (“vila real”, 17.2)

De cadeia para observação

- Partimos a cadeia pelos espaços, obtendo uma lista
- A última cadeia da lista dá a temperatura,
- O resto dá a cidade:

```
observationFromString :: String -> Observation  
observationFromString xs = (a, b)
```

```
  where
```

```
    w = words xs
```

```
    b = read (last w)
```

```
    a = unwords (init w)
```

A função last dá o último elemento de uma lista. A função init dá a parte inicial da lista, isto é, o que fica quando eliminamos o último elemento.

```
Main> observationFromString "leiria 17"  
("leiria",17.0)
```

17-01-20(Main>

words e unwords

- A função `words` parte uma cadeia pelos espaços (ou sequência de espaços), dando uma lista.
- A função `unwords` junta as cadeias de uma lista, inserindo um espaço entre cada duas.

```
Main> words "anibal antonio cavaco silva"  
["anibal","antonio","cavaco","silva"]
```

```
Main> words "aaa bbb"  
["aaa","bbb"]
```

```
Main> words "123456"  
["123456"]
```

```
Main> words ""  
[]
```

```
Main> words "    aaa    bbb    "  
["aaa","bbb"]
```

```
Main>
```

```
Main> unwords ["laranjas", "morangos", "uvas"]  
"laranjas morangos uvas"
```

```
Main> unwords ["sardinha"]  
"sardinha"
```

```
Main> unwords []  
""
```

```
Main> unwords ["aaa", "    ", "bbb"]  
"aaa    bbb"
```

```
Main> |
```

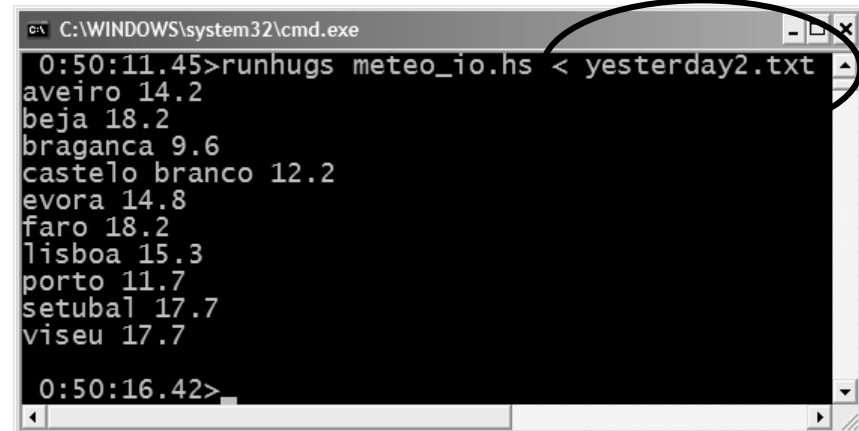
getObservations

- É simples:

```
getObservations :: IO Observations
getObservations =
  do
    s <- getContents
    return (map observationFromString (lines s))
```

- Exemplo: ordenar e mostrar na consola:

```
sortObservations :: IO ()
sortObservations =
  do
    x <- getObservations
    putObservations (qsort x)
```



```
C:\WINDOWS\system32\cmd.exe
0:50:11.45>runhugs meteo_io.hs < yesterday2.txt
aveiro 14.2
beja 18.2
braganca 9.6
castelo branco 12.2
evora 14.8
faro 18.2
lisboa 15.3
porto 11.7
setubal 17.7
viseu 17.7
0:50:16.42>
```

putObservations

- Primeiro a conversão de Observation para String:

```
stringFromObservation :: Observation -> String  
stringFromObservation (x,y) = x ++ " " ++ show y
```

- Para escrever as observações, primeiro convertemos cada observação para cadeia e depois juntamos todas as cadeias, inserindo newlines após cada cadeia:

```
putObservations :: Observations -> IO ()  
putObservations xs =  
    putStr (unlines (map stringFromObservation xs))
```

unlines

- A função `lines` parte uma cadeia pelos caracteres *newline*, dando uma lista de cadeias.
- A função `unlines` faz o contrário: dada uma lista de cadeias junta-as todas numa cadeias, cada uma finalizada por *newline*.

```
Main> unlines ["besugo", "dourada", "robalo", "salmonete"]  
"besugo\ndourada\nrobalo\nsalmonete\n"  
Main>
```


readFile, writeFile

- Para ler tudo de um ficheiro, especificado pelo seu pathname, usamos a função `readFile`.
- Para escrever uma cadeia num ficheiro usamos a função `WriteFile`.
- Observe os tipos destas funções:

```
Main> :type readFile
readFile :: FilePath -> IO String
Main> :type writeFile
writeFile :: FilePath -> String -> IO ()
Main>
```

O tipo `FilePath` é o mesmo que `String`. Representa as cadeias que são nomes de ficheiro.

Exemplo com readFile e writeFile

- Ler o ficheiro “yesterday1.txt”, ordenar por temperatura decendente e guardar em “out.txt”.

```
sortYesterdayFile :: IO ()
sortYesterdayFile =
  do
    s <- readFile "yesterday1.txt"
  let
    x = map observationFromString (lines s)
    y = qsortTempDown x
    z = unlines (map stringFromObservation y)
  in
    writeFile "out.txt" z
```

Cuidado
com o
alinhamento
do let!

Note bem: a ordem das
definições dentro do let é
irrelevante. Experimente
trocar.

Exercício: programar
isto sem usar o let.

Parametrizando o ficheiro

- Exemplo, um programa para consultar a tabela das observações, a qual terá sido lida do ficheiro indicado em argumento:

```
queryCitiesFromFile :: FilePath -> IO ()  
queryCitiesFromFile pName =  
  do  
    s <- readFile pName  
    queryCities (map observationFromString (lines s))
```

Interrogando a tabela

- Recorremos à função find:

```
queryCities :: Observations -> IO ()
```

```
queryCities xs =
```

```
  do
```

```
    s <- getLine
```

```
    if s /= "" then
```

```
      do
```

```
        print (find s xs)
```

```
        queryCities xs
```

```
    else
```

```
      return ()
```

```
Main> queryCitiesFromFile "temps2.txt"
```

```
aaa
```

```
[10.0]
```

```
xxx
```

```
[14.0]
```

```
bbb
```

```
[14.0]
```

```
ccc
```

```
[]
```

```
bbb
```

```
[14.0]
```

```
ttt
```

```
[9.0,10.0]
```

```
uuu
```

```
[]
```

```
ggg
```

```
[10.0,14.0,11.0,10.0]
```

```
Main> queryCitiesFromFile "yesterday2.txt"
```

```
faro
```

```
[18.2]
```

```
setuba1
```

```
[17.7]
```

```
funchal
```

```
[]
```

```
lisboa
```

```
[]
```

```
Main>
```

find em compreensão

- Palavras para quê?

```
find :: Eq a => a -> [(a, b)] -> [b]  
find k t = [v | (k', v) <- t, k == k']
```

Antes, programámos
assim ☹

```
find :: Eq a => a -> [(a, b)] -> [b]  
find _ [] = []  
find x ((a,b):ts)  
  | x == a = b : z  
  | otherwise = z  
  where  
    z = find x ts
```

Controlo

- Que faz a função lines? E a função unlines?
- Que faz a função words? E a função unwords?
- Podemos trocar as definições das variáveis locais dentro de um let ou de um where?
- Como é o find em compreensão?

Exercícios

- Programe por si as funções do prelúdio que usámos na aula: `lines`, `unlines`, `words`, `unwords`, `last`, `init`.
- Programe uma função que lê dois ficheiros com temperaturas de várias cidades e que, para cada cidade que existe nos dois ficheiros, escreve uma linha num ficheiro de saída, com o nome da cidade seguido de `+`, se a temperatura subiu, `=`, se se manteve e `-`, se desceu.

Na próxima aula

- Mudaremos de assunto, e começaremos a estudar as funções de ordem superior.

Fundamentos da Programação

Décima quarta aula:
Funções de ordem superior

Nesta aula vamos...

- Estudar as funções de ordem superior, em Haskell.
- Funções de ordem superior são funções em que um dos argumentos é uma função.
- Na verdade, não têm nada de especial.

Funções de ordem superior

- Sobre listas:
 - map
 - filter
 - foldr, foldl
- Há outras, que não são sobre listas.

map

- Aplica uma função dada a todos os elementos de uma lista.
- Vem no prelúdio, mas programar-se-ia assim:

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

```
mapList f xs = [f x | x <- xs]
```

Em compreensão

```
mapList' :: (a->b) -> [a] -> [b]
```

```
mapList' _ [] = []
```

```
mapList' f (x:xs) = f x : mapList' f xs
```

Recursivamente

Exemplos com map

- Das aulas anteriores:

```
getObservations =
```

```
...
```

```
    return (map observationFromString (lines s))
```

Com funções
definidas no
programa.

```
Main> map (+1) [10..13]
[11,12,13,14]
Main> map (>5) [3,5,7,9]
[False,False,True,True]
Main> map length [[2,4],[],[10..33]]
[2,0,24]
Main> map (`mod` 2) [5,13,4,1,8]
[1,1,0,1,0]
Main> map (*2) [8,1,9]
[16,2,18]
Main>
```

Com funções do
prelúdio e com
secções.

Expressões lambda

- As expressões lambda representam funções, sem lhes dar um nome.
- Por exemplo, a função que calcula o dobro de um número mais 1 pode ser definida pela expressão $\lambda x \rightarrow 2 * x + 1$.
- Nos computadores, como não há a letra lambda, usa-se a barra para trás:

```
Main> (\x -> 2*x + 1) 4
```

```
9
```

```
Main> (\s -> "<"+s++">") "garoupa"  
"<garoupa>"
```

```
18-01 Main>
```

Usam-se como argumento das funções de ordem superior

- Se consideramos que a transformação que vamos aplicar a todos os elementos da lista, por via da função `map`, não tem relevância que justifique ser programada explicitamente como uma função com nome, usamos uma expressão `lambda`:

```
Main> map (\x -> 2*x+1) [1..6]
[3,5,7,9,11,13]
Main> map (\x -> mod x 3 == 0) [2, 4..20]
[False,False,True,False,False,True,False,False,True,False]
Main> map (\s -> "<"+s++">") ["garoupa", "pargo", "tamboril"]
["<garoupa>", "<pargo>", "<tamboril>"]
Main> map (\x -> replicate x '-') [8,7..1]
["-----", "-----", "-----", "-----", "-----", "-----", "-----", "-----"]
Main> |
```

filter

- Selecciona de uma lista os elementos que satisfazem um predicado dado.
- Um predicado é uma função booleana, com um argumento.
- Vem no prelúdio, mas programar-se-ia assim:

Em compreensão

```
filterList :: (a->Bool) -> [a] -> [a]  
filterList p xs = [x | x <- xs, p x]
```

Recursivamente

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


Exemplos com filter

- A função warmer:

```
warmerThan :: Observation -> Observations -> Observations
warmerThan x ys = [y | y <- ys, snd y > snd x]
```

Em compreensão

```
warmerThan' :: Observation -> Observations -> Observations
warmerThan' x ys = filter (\y -> snd y > snd x) ys
```

Com expressão
lambda.

```
Main> warmerThan ("ttt",10) [("aaa",12),("zzz",8),("eee",10),("rrr",14)]
[("aaa",12.0),("rrr",14.0)]
Main> warmerThan' ("ttt",10) [("aaa",12),("zzz",8),("eee",10),("rrr",14)]
[("aaa",12.0),("rrr",14.0)]
Main> |
```

Quicksort com filtro 😐

- Concatenamos os que são menores do que a cabeça, depois de ordenados, com os que são iguais à cabeça, com os que são maiores do que a cabeça, depois de ordenados:

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

Quicksort com duas passagens

- Podemos poupar o filtro dos iguais, e assim melhorar a eficiência, realizando apenas duas passagens na lista, de cada vez:

```
qsort' :: Ord a => [a] -> [a]
qsort' [] = []
qsort' (x : xs) =
    qsort' (filter (<=x) xs) ++
    [x] ++
    qsort' (filter (>x) xs)
```

foldr

- Aplica uma função binária sucessivamente, à sequência de elementos de uma lista: cada resultado parcial é operado com o elemento seguinte.
- Exemplo: somar uma lista.
- Ao somar uma lista, cada soma parcial é somada com o elemento seguinte, constituindo uma nova soma parcial.
- Na verdade, a soma vem da direita para a esquerda, e cada elemento é somado com a soma parcial dos elementos mais à direita.

soma, produto, concatenação

- Estas três funções têm a mesma “forma”:

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

```
productList :: [Int] -> Int
productList [] = 1
productList (x:xs) = x * productList xs
```

```
concatList :: [String] -> String
concatList [] = ""
concatList (x:xs) = x ++ concatList xs
```

São análogas:
varia a
operação, (+),
(*) e (++), e o
valor inicial, 0,
1 e [].

Cálculo da soma

- Por exemplo:

```
sumList [3,8,1,4] =  
3 + sumList [8,1,4] =  
3 + (8 + sumList [1,4]) =  
3 + (8 + (1 + sumList [4])) =  
3 + (8 + (1 + (4 + sumList []))) =  
3 + (8 + (1 + (4 + 0))) =  
3 + (8 + (1 + 4)) =  
3 + (8 + 5) =  
3 + 13 =  
16
```

É como se a lista fosse sendo “dobrada” da direita para a esquerda, até sair o resultado.

Repare: a lista [3,8,1,4] pode escrever-se 3:(8:(1:(4:[]))). Logo, a expressão da soma é análoga à da lista, substituindo (:) por (+) e [] por 0.

foldr = “fold from the right”

Generalizando: foldr

- Observe:

Nos casos mais simples e mais habituais, os tipos a e b são o mesmo.

```
foldrList :: (a -> b -> b) -> b -> [a] -> b
foldrList _ z [] = z
foldrList op z (x:xs) = op x (foldrList op z xs)
```

- A última equação podia ser escrita usando o operador infixamente:

```
...
foldrList op z (x:xs) = x `op` foldrList op z xs
```

Repare: tal como na soma, as contas fazem-se da direita para a esquerda.

folds célebres

- sum
- product
- concat
- and
- or

```
sumList' :: Num a => [a] -> a  
sumList' xs = foldr (+) 0 xs
```

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

```
concatList' :: [[a]] -> [a]  
concatList' xs = foldr (++) [] xs
```

```
andList' :: [Bool] -> Bool  
andList' xs = foldr (&&) True xs
```

```
orList' :: [Bool] -> Bool  
orList' xs = foldr (||) False xs
```


Experimentando directamente

- Observe, usando a função foldr do prelúdio:

```
Main> foldr (+) 0 [1..10]
55
Main> foldr (+) 88 [1..10]
143
Main> foldr (*) 1 [1..10]
3628800
Main> foldr (*) 1000000 [1..10]
3628800000000
Main> foldr (++) [] ["aaa","bbb","ccc"]
"aaabbbccc"
Main> foldr (++) "zzz" ["aaa","bbb","ccc"]
"aaabbbccczzz"
Main> foldr (&&) True (map (\x -> mod x 3 == 0) [25, 50, 12, 8])
False
Main> foldr (||) False (map (\x -> mod x 3 == 0) [25, 50, 12, 8])
True
Main> |
```

Controlo

- Que funções de ordem superior vimos hoje?
- Qual é o tipo da função foldr?
- O que são expressões lambda?

Exercícios

- Programe uma função `mapIf` que aplica uma dada transformação a todos os elementos de uma lista que verificam um dado predicado, deixando os outros na mesma.
- Programe a função `takeWhile` que seleccionam da lista todos os elementos até aparecer um que não verifica o predicado dado, ou até chegar ao fim da lista.

Na próxima aula

- Estudaremos mais algumas funções de ordem superior.
- Algumas dessas já não terão a ver com listas.

Fundamentos da Programação

Décima quinta aula:
Listas infinitas

Nesta aula vamos...

- Continuar a estudar a função foldr.
- De passagem, encontrar algumas outras funções de ordem superior.
- Aprender a trabalhar com listas infinitas.

Experimentando o foldr

- Nos exemplos que vimos até agora, a função era comutativa.
- Vejamos um exemplo com uma função não comutativa:

```
Main> foldr (/) 1 [1,2,3]  
1.5
```

- Claro:

```
foldr (/) 1 [1,2,3] =  
1 / (2 / (3 / 1)) =  
1 / (2 / 3) =  
3 / 2 =  
1.5
```

Mais difícil ainda

```
Main> foldr (\x y -> y / x) 1 [1,2,3]  
0.16666666666666667
```

- Claro:

```
foldr (\x y -> y / x) 1 [1,2,3] =  
foldr (\x y -> y / x) 1 [2,3] / 1 =  
(foldr (\x y -> y / x) 1 [3] / 2) / 1 =  
((foldr (\x y -> y / x) 1 [] / 3) / 2) / 1 =  
((1 / 3) / 2) / 1 =  
...  
1 / 6 =  
0.16666666666666667
```

- $\text{foldr } (\backslash x y \rightarrow y / x) 1 [1..n]$ vale $1 / (\text{factorial } n)$

```
Main> map (\n -> foldr (\x y -> y / x) 1 [1..n]) [1..6]  
[1.0,0.5,0.16666666666666667,0.04166666666666667,0.008333333333333333,  
0.0013888888888888889]
```


flip

- flip é a função de ordem superior que troca os argumentos:

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

- Em vez de `\x y -> y / x` podemos escrever `flip (/)`:

```
Main> map (\n -> foldr (flip (/)) 1 [1..n])[1..6]
[1.0,0.5,0.1666666666666667,0.0416666666666667,0.008333333333333333,
0.0013888888888888889]
```

Máximo da lista de cadeias

- Já sabemos que basta usar a função `maximum`, do prelúdio.
- Mas poderia ser assim:

```
maximumList :: [String] -> String  
maximumList xs = foldr max "" xs
```

```
Main> maximumList ["lisboa", "coimbra", "porto", "aveiro"]  
"porto"  
Main> maximumList ["eee", "aaa", "ttt", "rrr", "bbb"]  
"ttt"  
Main> maximumList []  
""  
Main> maximumList ["sardinha"]  
"sardinha"
```

Cálculo do máximo da lista

- Recursivamente, pela definição de foldr:

```
foldr max "" ["lisboa", "coimbra", "porto", "aveiro"] =  
max "lisboa" (foldr max "" ["coimbra", "porto", "aveiro"]) =  
max "lisboa" (max "coimbra" (foldr max "" ["porto", "aveiro"])) =  
max "lisboa" (max "coimbra" (max "porto" (foldr max "" ["aveiro"]))) =  
max "lisboa" (max "coimbra" (max "porto" (max "aveiro" (foldr max "" [])))) =  
max "lisboa" (max "coimbra" (max "porto" (max "aveiro" ""))) =  
max "lisboa" (max "coimbra" (max "porto" "aveiro")) =  
max "lisboa" (max "coimbra" "porto") =  
max "lisboa" "porto" =  
"porto"
```

- Ou desdobrando, infixamente:

```
foldr max "" ["lisboa", "coimbra", "porto", "aveiro"] =  
"lisboa" `max` ("coimbra" `max` ("porto" `max` ("aveiro" `max` ""))) =  
...  
"porto"
```

E o mínimo da lista?

- Que tal?

```
minimumList :: [String] -> String  
minimumList xs = foldr min "" xs
```

```
Main> minimumList ["laranja", "banana", "morango"]  
""
```

```
Main> minimumList ["zzz", "eee", "sss", "hhh"]  
""
```

```
Main>
```

- Como a cadeia vazia é menor do que qualquer outra cadeia, ao calcular o mínimo entre cadeia vazia e outra dá sempre cadeia vazia.
- Logo, a cadeia inicial para o foldr deve ser a maior cadeia possível!

A máxima cadeia

- Se só usarmos letras minúsculas, a cadeia máxima será uma cadeia com muitos 'z'.
- Podemos consertar a função `minimumList` assim:

```
minimumList :: [String] -> String  
minimumList xs = foldr min (replicate 1000 'z') xs
```

```
Hugs> minimumList ["laranja", "banana", "morango"]  
"banana"  
Main> minimumList ["zzz", "eee", "sss", "hhh"]  
"eee"  
Main> |
```

Mas se a lista tiver uma cadeia que comece por mais de 1000 'z' o resultado estará errado ☹

foldr1

- O Haskell evita estas complicações introduzindo a função `foldr1`, aplicável a listas não vazias que usa o último elemento da lista com elemento inicial para a aplicação da função paramétrica:

```
foldr1List :: (a -> a -> a) -> [a] -> a
foldr1List _ [x] = x
foldr1List f (x:xs) = f x (foldr1List f xs)
```

```
minimumList :: [String] -> String
minimumList [] = error "minimumList: empty list"
minimumList xs = foldr1List min xs
```

Listas infinitas

- Na verdade, a maior cadeia só com letras minúsculas seria uma cadeia infinita só com zês.
- Como conseguir uma cadeia infinita em Haskell?

- É simples:

```
infiniteString :: Char -> String  
infiniteString x = x : (infiniteString x)
```

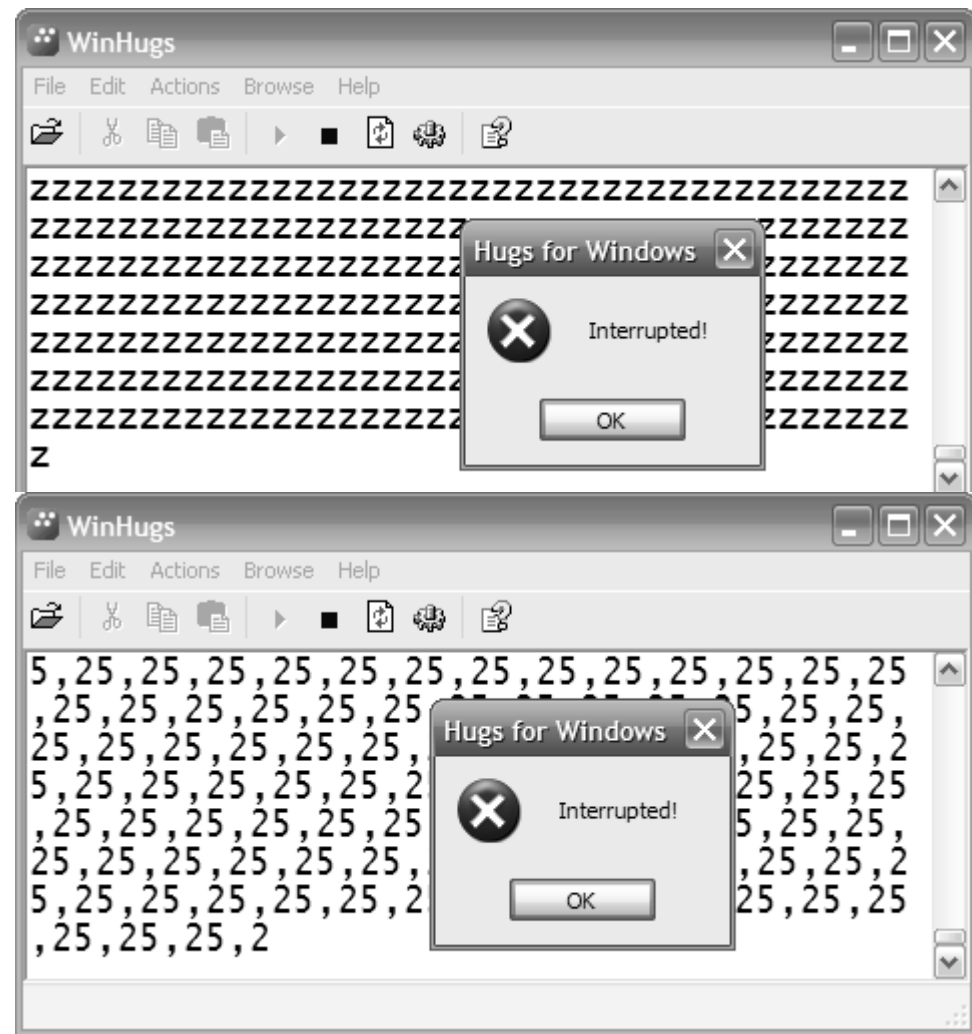
- Mais simples ainda, se for genérico:

```
infinite :: a -> [a]  
infinite x = x : (infinite x)
```

Isto no prelúdio
chama-se repeat.

Cuidado com as listas infinitas!

- Se chamarmos `infiniteString 'z'` no Hugs, nunca mais param os zês...
- Se chamarmos `infinite 25`, nunca mais param os 25!



Listas infinitas até certo ponto

- Não podemos calcular as listas infinitas até ao fim, claro.
- Mas pode nem ser preciso:

```
Main> take 25 (infiniteString 'z')  
"zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz"  
Main> replicate 25 'z'  
"zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz"  
Main> take 8 (infinite 25)  
[25,25,25,25,25,25,25,25]  
Main> zip [1..10] (repeat 32)  
[(1,32),(2,32),(3,32),(4,32),(5,32),(6,32),  
(7,32),(8,32),(9,32),(10,32)]  
Main> concat (take 10 (repeat "+**+"))  
"+**++**++**++**++**++**++**++**++**++**++**"
```

Mais listas infinitas

- A lista dos números naturais:

```
naturalsFrom :: Int -> [Int]
naturalsFrom x = x : naturalsFrom (x+1)
```

```
naturals :: [Int]
naturals = naturalsFrom 0
```

```
Hugs> take 10 naturals
[0,1,2,3,4,5,6,7,8,9]
```

```
progression :: Int -> Int -> [Int]
progression x y = x : progression (x+y) y
```

```
Main> take 10 (progression 10 3)
[10,13,16,19,22,25,28,31,34,37]
Main> take 10 (progression (-1) (-1))
[-1,-2,-3,-4,-5,-6,-7,-8,-9,-10]
```

Mas isto não é novidade!

- Na verdade, as funções da página anterior são desnecessárias, pois o seu comportamento pode ser obtido directamente:

```
Main> take 10 [1..]
[1,2,3,4,5,6,7,8,9,10]
Main> take 10 [10, 13..]
[10,13,16,19,22,25,28,31,34,37]
Main> take 10 [(-1), (-2)..]
[-1,-2,-3,-4,-5,-6,-7,-8,-9,-10]
```

Lista dos quadrados

- Podíamos definir assim:

```
squares' :: [Int]
squares' = map (^2) [0..]
```

```
Main> take 10 squares'
[0,1,4,9,16,25,36,49,64,81]
```

- Mas é mais interessante e melhor assim:

```
squares_ :: Int -> Int -> [Int]
squares_ x y = x : squares_ (x+y)(y+2)
```

```
squares :: [Int]
squares = squares_ 0 1
```

```
Main> take 10 squares
[0,1,4,9,16,25,36,49,64,81]
Main> take 8 (squares_ 10 1)
[10,11,14,19,26,35,46,59]
Main> take 8 (squares_ 5 5)
[5,10,17,26,37,50,65,82]
```

Números de Fibonacci

- A sucessão de Fibonacci [0,1,1,2,3,5,8,...] define-se assim:

```
fib :: Int -> Int -> [Int]
fib x y = x : (fib y (x+y))
```

```
fibonacci :: Int -> [Int]
fibonacci x = take x (fib 0 1)
```

```
Main> fibonacci 40
[0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181,6765,10946,17711,28657,46368,75025,121393,196418,317811,514229,832040,1346269,2178309,3524578,5702887,9227465,14930352,24157817,39088169,63245986]
```

- É bem melhor do que a definição elementar:

```
fibn :: Int -> Int
fibn 0 = 0
fibn 1 = 1
fibn (x+2) = fibn (x+1) + fibn x
```

```
fibonacci' :: Int -> [Int]
fibonacci' x = map fibn [0..x-1]
```

```
Main> fibonacci' 40
[0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181,6765,10946,17711,28657,46368,75025,121393,196418,317811,514229,832040,1346269,2178309,3524578,5702887,9227465,14930352,24157817,39088169,63245986]
```



Controlo

- Para que serve o flip?
- Quanto tempo leva a calcular uma lista infinita?
- O que são números de Fibonacci.
- Quanto vale sum naturals?

Exercícios

- Programe o máximo divisor comum de uma lista de números, usando um foldr.
- Programe a sucessão infinita $n \rightarrow n^2 + 5$.
- Programe a sucessão infinita das potências de x .
- Programe a sucessão infinita dos cubos perfeitos
- Programe a sucessão de Fibonacci de terceira ordem, em que cada elemento é a soma dos três anteriores.

Na próxima aula

- Continuaremos as nossas experiências com as funções de ordem superior e com as listas infinitas.

Fundamentos da Programação

Décima sexta aula:
Mais sobre funções de ordem
superior

Nesta aula vamos...

- Rever alguns pontos da aula anterior
- Estudar o foldl.
- Aprender a técnica dos parâmetros de acumulação.

Os quadrados

- A lista infinita dos quadrados perfeitos, de novo:

```
squares_ :: Int -> Int -> [Int]
squares_ x y = x : squares_ (x+y)(y+2)

squares :: [Int]
squares = squares_ 0 1
```

- Cálculo infinito:

```
squares =
squares_ 0 1 =
0 : squares_ 1 3 =
0 : (1 : squares 4 5) =
0 : (1 : (4 : squares 9 7)) =
0 : (1 : (4 : (9 : squares 16 9))) =
...
```

Os números de Fibonacci

- A lista infinita dos números de Fibonacci.

```
fib :: Int -> Int -> [Int]
fib x y = x : fib y (x+y)
```

```
fibs :: [Int]
fibs = fib 0 1
```

- Cálculo infinito:

```
fibs =
fib 0 1 =
0 : fib 1 1 =
0 : (1 : fib 1 2) =
0 : (1 : (1 : fib 2 3)) =
0 : (1 : (1 : (2 : fib 3 5))) =
0 : (1 : (1 : (2 : (3 : fib 5 8)))) =
0 : (1 : (1 : (2 : (3 : (5 : fib 8 13))))) =
```

...

A função flip

- Programando com expressões lambda:

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

- Programando “à moda antiga”:

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

- Ou mesmo:

```
flip_ :: (a -> b -> c) -> (b -> a -> c)  
flip_ f = g  
  where g x y = f y x
```

Os bons e os maus

- Colocar os elementos *bons* de uma lista antes dos *maus*.
- Ser *bom* é satisfazer um predicado paramétrico p ; ser *mau* é não satisfazer ou, o que é o mesmo, satisfazer o predicado $\lambda x \rightarrow \text{not } (p\ x)$, para o p dado.

```
goodBefore :: (a -> Bool) -> [a] -> [a]
goodBefore p xs = filter p xs ++ filter q xs
  where q x = not (p x)
```

- Ou, com expressões lambda:

```
goodBefore :: (a -> Bool) -> [a] -> [a]
goodBefore p xs = filter p xs ++ filter (\x -> not (p x)) xs
```

```
Main> goodBefore even [1..10]
[2,4,6,8,10,1,3,5,7,9]
```

```
Main> goodBefore (\x -> mod x 3 == 0) [1..10]
[3,6,9,1,2,4,5,7,8,10]
```

Parâmetros de acumulação

- Recordando a soma, versão básica:

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

```
sumList [3,7,2]=
3 + sumList [7,2] =
3 + (7 + sumList [2]) =
3 + (7 + (2 + sumList [])) =
3 + (7 + (2 + 0)) =
12
```

- Com parâmetro de acumulação:

```
sum_ :: Num a => [a] -> a
sum_ xs = sum' 0 xs

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

```
sum_ [3,7,2]=
sum' 0 [3,7,2] =
sum' (0+3) [7,2] =
sum' 3 [7,2] =
sum' (3+7) [2] =
sum' 10 [2] =
sum' (10+2) [] =
sum' 12 [] =
12
```

Nova inversão

- Inverter uma lista, antigamente:

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

Recorde que se trata de uma função relativamente ineficiente, de complexidade quadrática.

- Com parâmetro de acumulação:

```
reverse_ :: [a] -> [a]
reverse_ xs = rev' [] xs
```

```
rev' :: [a] -> [a] -> [a]
rev' vs [] = vs
```

```
rev' vs (x:xs) = rev' (x:vs) xs
```

```
reverse_ [3,7,2,8] =
rev' [] [3,7,2,8] =
rev' [3] [7,2,8] =
rev' [7,3] [2,8] =
rev' [2,7,3] [8] =
rev' [8,2,7,3] [] =
[8,2,7,3]
```

Esta é bem melhor: complexidade linear.

Função mistério

- Que calcula a seguinte função, que também tem um parâmetro de acumulação?

```
mystery :: [Int] -> Int
mystery xs = dv 0 xs

dv :: Int -> [Int] -> Int
dv v [] = v
dv v (x:xs) = dv (10*v+x) xs
```

Experimentando e calculando

```
Main> mystery [5,1,8,4]
5184
Main> mystery [1]
1
Main> mystery [7,6..1]
7654321
```

```
dv :: Int -> [Int] -> Int
dv v [] = v
dv v (x:xs) = dv (10*v+x) xs
```

```
mystery [5,1,4,8] =
dv 0 [5,1,4,8] =
dv (10*0+5) [1,4,8] =
dv 5 [1,4,8] =
dv (10*5+1) [4,8] =
dv 51 [4,8] =
dv (51*10+4) [8] =
dv 514 [8] =
dv (514*10+8) [] =
dv 5148 [] =
5148
```

- Conclusão: se os números da lista forem dígitos, a função calcula o número decimal correspondente.

Recapitulando

- As três funções, sum' , rev' e dv seguem o mesmo padrão:
- Quando a lista é vazia, o resultado é o acumulador.
- Quando não é, aplica-se a função recursivamente, ao acumulador, já modificado por uma função binária, e ao resto da lista.
- Essa função binária é o que muda de umas para as outras.

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

```
rev' :: [a] -> [a] -> [a]
rev' v [] = v
rev' v (x:xs) = rev' (x:v) xs
```

```
dv :: Int -> [Int] -> Int
dv v [] = v
dv v (x:xs) = dv (10*v+x) xs
```

Funções de acumulação

- Para a soma, é (+)
- Para a inversão, é $\lambda x y \rightarrow y : x$
- Para a função dv é $\lambda x y \rightarrow 10 * x + y$

```
Main> (+) 4 7
```

```
11
```

```
Main> (\x y -> y : x) [5,7,2] 8
```

```
[8,5,7,2]
```

```
Main> (\x y -> 10 * x + y) 3 7
```

```
37
```

```
Main> (\x y -> 10 * x + y) 344 701
```

```
4141
```

foldl

- O padrão das três funções é o do foldl:

```
foldlList :: (a -> b -> a) -> a -> [b] -> a
foldlList _ z [] = z
foldlList op z (x:xs) = foldlList op (op z x) xs
```

```
sum_a :: Num a => [a] -> a
sum_a xs = foldl (+) 0 xs
```

```
reverse_a :: [a] -> [a]
reverse_a xs = foldl (flip (:)) [] xs
```

```
mystery_a :: [Int] -> Int
mystery_a xs = foldl (\x y -> 10 * x + y) 0 xs
```

```
Main> sum_a [1..8]
36
Main> reverse_a "tttyyyhhh123"
"321hhhyyyttt"
Main> mystery_a [5,1,1,1,6,8]
511168
```

NB: $\backslash x y \rightarrow y : x$ é o mesmo que $\text{flip } (:)$.

Controlo

- Qual é a diferença entre foldr e foldl?
- Como será o foldl1?
- Qual é mais eficiente, o reverse antigo ou o novo, programado com parâmetro de acumulação?
- Experimente trocar o foldl pelo foldr nos exemplos de hoje. Que tal?
- Refaça os cálculos dos exemplos de hoje usando directamente a definição de foldl.

Exercícios

- Programe o comprimento com parâmetros de acumulação
- Idem, para o produto.
- Idem, para o factorial.
- E depois, use o foldl, para unificar essas funções todas.
- Use o foldl para converter uma cadeia numérica para o número representado. Use a função digitToInt, que converte um algarismo para o seu valor inteiro. Por exemplo digitToInt '4' vale 4.

Na próxima aula

- Vamos ajudar o professor a programar as funções para calcular as notas da nossa cadeira, de acordo com as regras de avaliação.

Fundamentos da Programação

Décima sétima aula: Problema das notas

Nesta aula vamos...

- **Começar a programar o cálculo das notas da nossa cadeira.**
- **Assim ilustraremos o processo de resolução de problemas de programação.**
- **E, enquanto programamos, reencontraremos algumas técnicas de programação importantes.**

Onde estão os dados?

- **Há uma listagem dos alunos inscritos.**
- **Por cada concurso no Mooshak há um ficheiro com o ranking.**
- **Por cada questionário há um ficheiro com notas de todas as respostas de cada estudante.**
- **Há um ficheiro do Moodle com as notas de todos os relatórios.**

Quais são os resultados

- **Queremos uma pauta dos questionários, para contar os pontos obtidos por cada estudantes.**
- **Queremos uma pauta dos guiões e problemas, incluindo as notas dos relatórios.**
- **Queremos uma pauta dos exercícios separada.**
- **Nestas três pautas, os alunos são identificados pelo número.**
- **A listagem dos alunos permite descodificar.**

O Excel ajudará

- **Os quatro ficheiros que produziremos – listagem dos alunos, pauta dos questionários, pauta dos guiões e problemas, pauta dos exercícios – constituirão folhas de um livro no Excel, com o qual faremos as contas finais.**
- **Haverá mais tarde novas folhas para os resultados dos exames.**

O ficheiro dos alunos

- É obtido nos serviços académicos.
- Tem quatro campos: número de ordem, número de aluno, nome completo e curso.
- Os campos estão separados por *tab*.
- Queremos apenas o número e nome.
- Tarefa: ler o ficheiro e produzir uma lista de pares número – nome.

Aspecto do ficheiro

Observe os *tabs* e as mudanças de linha.

```
68 68→34734>GUSTAVO · OLIVEIRA · SANTOS>LEI CRLF
69 69→19091>HELDER · LUÍS · GOMES · ALCARVA>LEI CRLF
70 70→34736>HÉLIO · ADRIANO · DOS · SANTOS · CORREIA→LEI CRLF
71 71→25079>HENRIQUE · PEDRO · DUARTE>LEI CRLF
72 72→31745>HUGO · MIGUEL · AVELINO · GASPAR→LEI CRLF
73 73→10389>HUGO · MIGUEL · DOS · SANTOS · MESTRE>LEI CRLF
74 74→16860>HUGO · MIGUEL · PEREIRA · DA · CRUZ · DENTA>LEI CRLF
75 75→31762>HUMBERTO · MIGUEL · LUÍS · CORREIA→LEI CRLF
76 76→17126>INÁCIO · JOSÉ · GONÇALVES · HORTA>LEI CRLF
77 77→34737>INÊS · SUSTELO · RIO · TORGAL · MENDES→LEI CRLF
78 78→33350>ISAURA · DENISE · FILIPE · CUAMBE>LEI CRLF
79 79→25080>IVO · ALEXANDRE · DE · OLIVEIRA · VARELA · CORREIA→LEI CRLF
80 80→31746>IVO · ALEXANDRE · GAGO · VTEGAS>LEI CRLF
```

Normal text nb char : 8141 Ln : 1 Col : 1 Sel : 0 Dos\Windows ANSI INS

- Talvez convenha eliminar os acentos e cedilhas nos nomes...

Separando os campos

- **Seria bom separar os campos usando a função words.**
- **Mas os nomes têm espaços.**
- **Solução: substituir os espaços por sublinhados, usar o words, escolher os campos que interessam, voltar a substituir os sublinhados nos nomes por espaços.**
- **Deixamos a questão dos acentos e cedilhas para depois.**

Substituindo caracteres

- Programemos uma função que substitui todas as ocorrências de um valor numa lista por um outro valor:

```
replace :: Eq a => a -> a -> [a] -> [a]
```

```
replace x y zs = map (\x' -> if x' == x then y else x') zs
```

```
Main> replace 3 7 ([1..5] ++ [7,5,3,1])  
[1,2,7,4,5,7,5,7,1]
```

```
Main> replace 'a' 'x' "a1mada"  
"x1mxdx"
```

Antes de conhecermos o map e as expressões lambda, programaríamos assim:

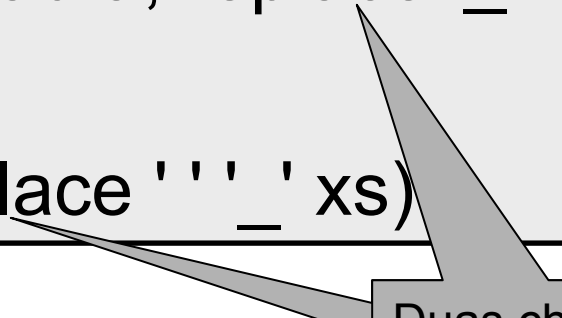
```
replace' :: Eq a => a -> a -> [a] -> [a]  
replace' x y [] = []  
replace' x y (z:zs)  
  | x == z = y : replace' x y zs  
  | otherwise = z : replace' x y zs
```

Processando a linha

- **Observe:**

```
type Student = (Int, String)
```

```
studentFromLine :: String -> Student  
studentFromLine xs = (read a, replace '_' ' ' b)  
  where  
    [_ ,a,b,_] = words (replace ' ' '_' xs)
```



Duas chamadas
de replace.

Criando a lista de pares

- **Lemos com `readFile`; partimos em linhas, com `lines`; processamos as linhas com `studentFromLine`; criamos a lista com `map`:**

```
readStudents :: FilePath -> IO [Student]
readStudents f =
  do
    s <- readFile f
    return (map studentFromLine (lines s))
```

Lendo e escrevendo

- Para controlar as operações:

```
testReadStudents :: IO ()
testReadStudents =
  do
    s <- readStudents "alunos_1.txt"
    print s
```

Este é um
ficheiro mais
pequeno, para
experimentar

Realmente, os
acentos ficam
mal no Winhugs



```
WinHugs
File Edit Actions Browse Help
[Icons]
Main> testReadStudents
[(33362,"ADELINA AUGUSTO JACINTO NHANALA"),
 (35010,"ALBANO MANUEL LEITE DE OLIVEIRA"),
 (34725,"ANA REIS DA SILVA LEAL"), (25071,"ANA RITA
CORREIA TEIXEIRA"), (24320,"ANDR\201 EDUARDO CARRI
\1990 RODRIGUES"), (34897,"ANDR\201 FILIPE DA FONSECA
CASTRO"), (31737,"ANDR\201 ISRAEL DANTAS DA ROCHA"),
 (30917,"ANDR\201 MANUEL CAVACO RAMOS"), (27973,"ANDR
\201 MANUEL GASPAR MORENO"), (20707,"ANDREIA DO CARMO
```

Eliminando os diacríticos

- Queremos substituir os A com acento ou til por A; idem para as outras vogais; e os Ç por C; e os Ñ por N.
- Primeiro uma função para substituir vários por um:

```
replaceThese :: Eq a => [a] -> a -> [a] -> [a]
replaceThese xs y zs =
    map (\x -> if elem x xs then y else x) zs
```

```
Main> replaceThese "aeiou" '#' "lisboa"
"l#sb###"
```

Eliminando todos

- Podemos fazer eliminações sucessivas:

```
removeDiacritics :: String -> String
```

```
removeDiacritics xs = replaceThese "ÁÀÃÂÄ" 'A' a
```

```
  where
```

```
    a = replaceThese "ÉÊÊË" 'E' e
```

```
    e = replaceThese "ÍÎÏ" 'I' i
```

```
    i = replaceThese "ÓÔÕÖ" 'O' o
```

```
    o = replaceThese "ÚÛÜ" 'U' u
```

```
    u = replace 'Ç' 'C' c
```

```
    c = replace 'Ñ' 'N' xs
```

```
Main> removeDiacritics "ÓSCAR GONÇALVES"  
"OSCAR GONCALVES"
```

```
Main> removeDiacritics "FÁTIMA SIMÕES"  
"FATIMA SIMOES"
```

```
Main> removeDiacritics "JOSÉ JÚLIO ESTÊVÃO"  
"JOSE JULIO ESTEVAO"
```

Removendo caracteres

- Já que estamos com a mão na massa, programemos uma função que elimina de uma lista todas as ocorrências de algum dos elementos de outra.
- Tem de ser parecida com a função `replaceThese`:

```
removeThese :: Eq a => [a] -> [a] -> [a]
```

```
removeThese xs ys = filter (\x -> not (elem x xs)) ys
```

```
Main> removeThese "abcde" "albufeira"  
"lufir"
```

```
Main> removeThese [0..9] [6,14,12,7,10,9,0,19]  
[14,12,10,19]
```

Controlo

- **Quais são os sinais diacríticos usados em português?**
- **A função read serve para quê?**
- **Qual é a diferença estrutural entre as funções replaceThese e removeThese?**
- **Qual é a assinatura da função readStudents? Que significa?**

Exercícios

- Programe a função `removeDiacritics` de maneira a fazer as operações numa só passagem na cadeia.
- Programe uma função `replaceMapped`, do género da `replaceThese` com dois argumentos de tipo lista, com o mesmo comprimento, tal que cada ocorrência de um elemento da primeira lista é substituída pelo correspondente valor na segunda lista. Por exemplo `replaceMapped “abc” “xyz” “braga”` vale `“yrxgx”`.
- Programe a função `removeDiacritics` usando a função `replaceMapped`, e trate logo do caso das letras minúsculas também.

Na próxima aula

- **Continuaremos a resolver o problema das notas.**
- **Analisaremos o caso das funções com vários argumentos.**
- **Veremos a composição de funções, que é uma função de ordem superior.**

Fundamentos da Programação

Décima oitava aula:
Processando os questionários

Nesta aula vamos...

- Tratar do processamento dos ficheiros dos questionários.
- É parecido com o do ficheiro dos estudantes, mas aparecem questões novas, interessantes.
- É outra vez um belo exemplo de programação.

Eis o ficheiro dos questionários

```
98 Humberto Miguel Correia>" 18 Janeiro 2008, 15:16">1 minuto 11 segundos->8CRLF
99 Humberto Miguel Correia>" 18 Janeiro 2008, 15:18">57 segundos->9CRLF
100 Ricardo José Adanjo Belchior->" 19 Janeiro 2008, 12:19">7 minutos 18 segundos->7CRLF
101 Ricardo José Adanjo Belchior->" 19 Janeiro 2008, 12:41">5 minutos 25 segundos->8CRLF
102 Ricardo José Adanjo Belchior->" 19 Janeiro 2008, 12:47">2 minutos 16 segundos->9CRLF
103 André Israel Dantas Rocha>" 19 Janeiro 2008, 12:29">9 minutos 26 segundos->10CRLF
104 Anthony Rodrigues Guerreiro>" 18 Janeiro 2008, 18:07">2 minutos 29 segundos->9.8CRLF
105 Daniel Nascimento Cadete->" 19 Janeiro 2008, 20:28">5 minutos->7.6CRLF
106 Daniel Nascimento Cadete->" 19 Janeiro 2008, 20:33">1 minuto 43 segundos->9.6CRLF
107 Dinis Pedro Pinto Madeira>" 18 Janeiro 2008, 14:26">8 horas 27 minutos->9CRLF
108 João Miguel Costa>" 18 Janeiro 2008, 18:51">7 minutos 17 segundos->7.8CRLF
109 João Miguel Coelho Higino>" 19 Janeiro 2008, 12:10">8 minutos 19 segundos->8CRLF
110 João Miguel Coelho Higino>" 19 Janeiro 2008, 12:21">3 minutos 3 segundos->8CRLF
111 João Miguel Coelho Higino>" 19 Janeiro 2008, 12:25">1 minuto 12 segundos->10CRLF
112 João Pedro Gomes->" 18 Janeiro 2008, 21:42">3 minutos 57 segundos->10CRLF
113 Tiago Miguel Baptista>" 19 Janeiro 2008, 15:56">4 minutos 31 segundos->10CRLF
114 Marco Bruno Santos Postana->" 18 Janeiro 2008, 18:18">0 minutos 30 segundos->8CRLF
```

Análise

- Neste caso queremos seleccionar a primeira coluna e a última (que é a quarta).
- Mas na primeira coluna, o nome não está em maiúsculas e, vendo bem, nem sequer é o nome completo. ☹
- Para cada linha queremos produzir um par *número de aluno – pontos*.
- Temos de ir buscar o número de aluno à tabela dos estudantes, procurando pelo nome.
- Os pontos valem 1 se a nota na última coluna for ≥ 7.0 , e 0 se não.

Continuação da análise

- Para cada aluno, só queremos o número de pontos mais elevado (que será 0 ou 1) uma vez.
- Todos os registos de um mesmo aluno parecem vir de seguida, mas pode haver excepções e temos de jogar pelo seguro.
- Não deve haver alunos que não figurem na lista de alunos.
- Mas será que há alunos com nomes ambíguos?

Estratégia

- Primeiro criamos uma lista de pares nome – nota.

```
type QuizResult = (String, Double)
```

- Precisamos de uma função para converter uma linha para um objecto QuizResult.

```
quizResultFromLine :: String -> QuizResult  
quizResultFromLine xs = ...
```

- É o esquema habitual...

De linha para QuizResult

- Podemos usar a técnica dos sublinhados ou, então, ir buscar o nome até ao primeiro *tab* e ir buscar o número à última das words.
- A palavra é colocada logo em maiúsculas, sem diacríticos.

```
quizResultFromLine :: String -> QuizResult
quizResultFromLine xs = (c, d)
  where
    a = takeWhile (/= chr 9) xs
    b = map toUpper a
    c = removeDiacritics b
    d = read (last (words xs))
```

takeWhile

- Dado um predicado e uma lista, calcula a lista formada pelo troço inicial da lista argumento até aparecer o primeiro elemento que não satisfaz o predicado, ou até ao fim da lista, se todos satisfizerem:

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

É uma função de ordem superior.

```
Main> takeWhile even [2,4,6,5,4,7]
[2,4,6]
```

```
Main> takeWhile (\x -> elem x "aeiou") "aeroporto"
"ae"
```

```
Main> takeWhile (/=0) [8,3,1,4,0,3,0,1,9]
[8,3,1,4]
```

toUpper

- toUpper x é a letra maiúscula correspondente à letra x.
- Para usar toUpper é preciso “importar” o módulo Char

```
Main> toUpper 't'
'T'
Main> toUpper 'F'
'F'
Main> toLower 'á'
'\225'
```

```
import Char
```

Logo no início
do ficheiro.

- Outras funções sobre caracteres:
 - isLower
 - isUpper
 - isAlpha
 - isDigit
 - isSpace
 - digitToInt
 - intToDigit
 - toLower
 - toUpper
 - ord
 - chr

toUpper, para cadeias

- Para colocar uma cadeia em maiúsculas, mapeia-se com toUpper:

```
Main> map toUpper "faro"
"FARO"
Main> map toUpper "albufeira"
"ALBUFEIRA"
Main> map toUpper "olhão"
"OLH\1950"
```

- Exercício: colocar uma cadeia com maiúscula inicial e o resto em minúsculas:

```
stringUpper1 :: String -> String
```

```
stringUpper1 [] = []
```

```
stringUpper1 (x:xs) = toUpper x : map toLower xs
```

```
Main> stringUpper1 "lisboa"
"Lisboa"
Main> stringUpper1 "aLbUfEiRa"
"Albufeira"
```

Correspondência de nomes

- No ficheiro das notas dos questionários, os nomes vêm abreviados.
- Com esses nomes incompletos, temos de ir buscar os números de alunos?
- Será que é possível?
- Haverá ambiguidades?
- O problema é: dado um nome incompleto, calcular o número do aluno.

Nomes incompletos

- As palavras do nome incompleto vêm pela ordem certa. Isto é, as palavras do nome incompleto constituem uma sublista da lista das palavras do nome completo.
- Precisamos de uma função para ver se uma lista é sublista de outra:

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

```
isSublist [] _ = True
```

```
isSublist _ [] = False
```

```
isSublist (x:xs) (y:ys)
```

```
  | x == y = isSublist xs ys
```

```
  | otherwise = isSublist (x:xs) ys
```

```
Main> isSublist [4,6,1,5] [3,4,5,6,1,3,5]
```

```
True
```

```
Main> isSublist "cao" "macaco"
```

```
True
```

```
Main> isSublist "gato" "sapato"
```

```
False
```

Busca por nome incompleto

- Temos de prever o caso de não haver correspondência ou de haver ambiguidade.
- De resto, é uma função de busca normal:

```
findNumber :: String -> [Student] -> Int
findNumber v t
  | null a = error ("findNumber: not found: " ++ v)
  | not (null (tail a)) = error ("findNumber: ambiguous: "
    ++ v ++ " " ++ show a)
  | otherwise = head a
  where a = [k | (k, s) <- t, isSublist v s]
```

Cálculo dos pontos

- Os pontos são o valor; a chave é o número de aluno:

```
type Quiz = (Int, Int)
```

- Passamos de QuizResult para Quiz com a ajuda da tabela de alunos:

```
quizFromQuizResult :: [Student] -> QuizResult -> Quiz
quizFromQuizResult t (x,y) =
    (findNumber x t, fromEnum (y >= 7.0))
```


Leitura

- Lemos o ficheiro dos questionários quando a lista dos alunos já está criada:

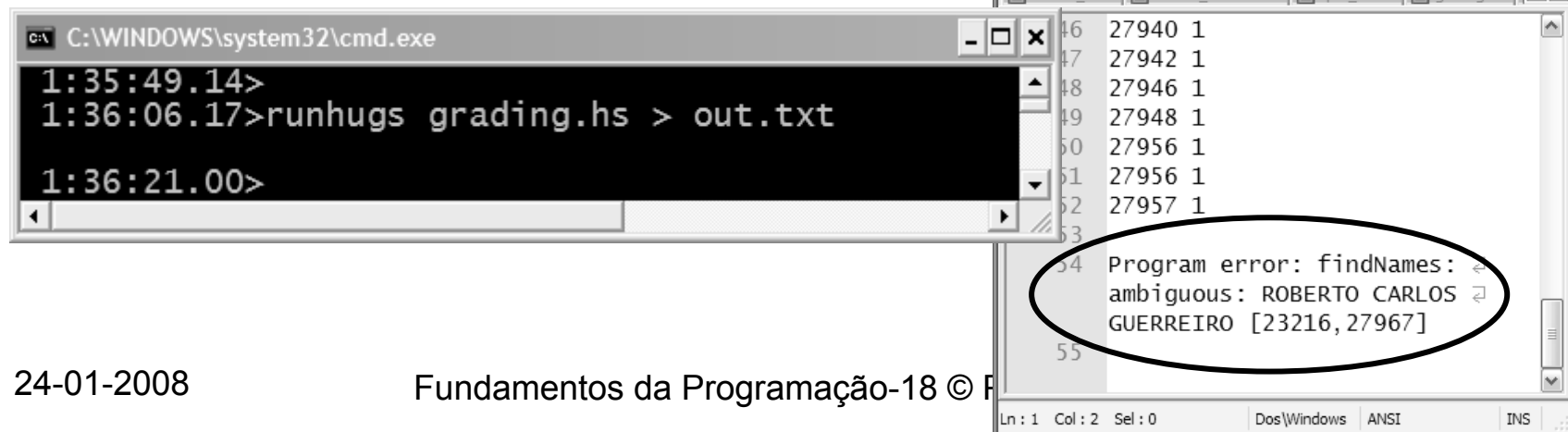
```
readQuizzes :: [Student] -> FilePath -> IO [Quiz]
readQuizzes t f =
  do
    s <- readFile f
    return (map (quizFromQuizResult t)
              (map quizResultFromLine (lines s)))
```

- Note bem: (quizFromQuizResult t) é a função que transforma um QuizResult num Quiz, com base na tabela de alunos t.

Experimentando

- Lemos os dois ficheiros e criamos a lista dos pontos, após ter criado a lista dos alunos:

```
testReadQuizzes :: IO ()
testReadQuizzes =
  do
    s <- readStudents "alunos_FP0708.txt"
    q <- readQuizzes s "quiz_14.txt"
    putStr (unlines (map \(a,b) -> show a ++ " " ++ show b) q))
```



Conclusão

- O processamento está OK, excepto que há pelo menos um caso de ambiguidade.
- Que fazer?
- Identificar todos os casos de ambiguidade e introduzir uma correcção, automaticamente.
- Essa correcção substituirá o nome ambíguo pelo nome completo, caso a caso, e volta a chamar a função findNumbers.
- Depois, finalmente, escolheremos a melhor nota (que será 1 ou 0...) para cada aluno.

Controlo

- Importamos o quê para usar as funções sobre caracteres?
- Há o `takeWhile`. Também há o `dropWhile`. Como será o `dropWhile`?
- Para que serve a função `chr`? E a função `ord`?
- Como podemos resolver as ambiguidades?
- Como fazemos para incluir na mensagem de erro da função `error` informação adicional variável, sobre as circunstâncias em que o erro ocorreu?
- Temos usado a função `fromEnum` para quê?

Exercícios

- Programe uma função para verificar se uma função é um prefixo (isto é, um troço inicial) de outra. Exemplo: “alga” é um prefixo de “algarve”.
- Idem, para verificar se é um sufixo. Exemplo “feira” é um sufixo de “albufeira”
- Idem para verificar se é subcadeia de outra. Uma subcadeia é uma sublista em que todos os elementos são contíguos. Exemplo “tolo” é uma subcadeia de “bartolomeu”.

Na próxima aula

- Processaremos os ficheiros que contêm os *rankings* do Mooshak.

Fundamentos da Programação

Décima nona aula:
Eliminação dos nomes ambíguos e
outros assuntos

Nesta aula vamos...

- Completar a programação do processamento dos questionários.
- Primeiro tratamos o caso dos nomes ambíguos.
- Depois guardaremos para cada aluno apenas a melhor nota.
- A propósito, estudaremos outros algoritmos interessantes.

Detectando as ambiguidades

- Uma pequena modificação no algoritmo de busca por nome incompleto permite detectar todos os casos problemáticos.
- Primeiro, escrevemos a função que busca os números de todos os estudantes com nome compatível:

```
findAllNumbers :: String -> [Student] -> [Int]  
findAllNumbers v t = [k | (k, s) <- t, isSublist v s]
```

Listando as ambiguidades

- Lemos os ficheiros e calculamos a função findAllNumbers para todos os nomes incompletos, filtrando depois não tiverem exactamente apenas um nome compatível.

```
testForProblems :: IO ()
testForProblems =
  do
    s <- readStudents "alunos_FP0708.txt"
    q <- readQuizResults s "quizz_14.txt"
  let
    n :: [String]
    n = fst (unzip q)
    m :: [(String, [Int])]
    m = [(x, findAllNumbers x s) | x <- n]
    p :: [(String, [Int])]
    p = filter \(_,xs) -> not (null xs) && not (null (tail xs)) m
  in
    print p
```

```
Char> testForProblems
[("ROBERTO CARLOS GUERREIRO",[23216,27967]),
 ("ROBERTO CARLOS GUERREIRO",[23216,27967]),
 ("JOAO PALMA",[31747,27956])]
```

Tabelas das correcções

- Associamos a cada nome incompleto ambíguo o nome completo, numa lista de pares:

```
corrections :: [(String, String)]  
corrections = [("ROBERTO CARLOS GUERREIRO",  
               "ROBERTO CARLOS GUERREIRO PINTO"),  
               ("JOAO PALMA", "JOAO FILIPE MEDEIRO DA PALMA")]
```

- Depois, quando houver ambiguidade, substituímos o nome incompleto pelo completo e repetimos.

Repetindo a consulta

- Repetimos a consulta com o nome completo, garantindo que a ambiguidade não se repete:

```
findNumber' :: String -> [Student] -> Int
findNumber' v t
  | null a = error ("findNames: not found: " ++ v)
  | not (null (tail a)) = findNumber' (head f) t
  | otherwise = head a
  where
    a = [k | (k, s) <- t, isSublist v s]
    f = find v corrections
```

```
quizFromQuizResult :: [Student] -> QuizResult -> Quiz
quizFromQuizResult t (x,y) = (findNumber' x t, fromEnum (y >= 7.0))
```

Escolhendo o último de cada

- Ordenamos a lista de pares de inteiros e escolhemos o último de cada grupo com a mesma chave:

```
choose :: [Quiz] -> [Quiz]
choose [] = []
choose [x] = [x]
choose (x1:x2:xs)
  | fst x1 == fst x2 = choose (x2:xs)
  | otherwise = x1 : choose (x2:xs)
```

```
Main> choose [(5,1),(5,12),(5,3),(6,2),(6,8),
(3,9),(4,1),(4,1),(4,9)]
[(5,3),(6,8),(3,9),(4,9)]
Main> choose [(5,1)]
[(5,1)]
Main> choose []
[]
```

Juntando tudo

- Retocamos a função readQuizzes:

```
readQuizzes' :: [Student] -> FilePath -> IO [Quiz]
readQuizzes' t f =
  do
    s <- readFile f
    let
      a = map (quizFromQuizResult t) (map quizResultFromLine (lines s))
      b = choose (qsort a)
    in
      return b

testReadQuizzes :: IO ()
testReadQuizzes =
  do
    s <- readStudents "alunos_FP0708.txt"
    q <- readQuizzes' s "quiz_14_part.txt"
    putStr (unlines (map \(a,b) -> show a ++ " " ++ show b) q))
```

```
Main> testReadQuizzes
17137 1
23013 1
23668 1
25089 1
25093 0
26133 1
27973 1
27976 0
27978 1
```

Problema de sexta-feira

- Contar as ocorrências numa lista dada de cada um dos valores presentes na lista.
- Por exemplo:

```
Main> countAll [3,4,3,4,3,4,5,6,7,7,6,6,6,6]
[(3,3),(4,3),(5,1),(6,6),(7,2)]
Main> countAll [8]
[(8,1)]
Main> countAll []
[]
Main> countAll "albufeira"
[('a',2),('l',1),('b',1),('u',1),('f',1),('e',1),('i',1),('r',1)]
Main> countAll ["aaa", "bbb", "aaa", "aaa", "ccc", "aaa", "ccc"]
[("aaa",4),("bbb",1),("ccc",2)]
Main> |
```

- Este é um problema clássico. Temos de saber resolvê-lo sem hesitar.

Solução recursiva

- Primeiro: uma função para contar o número de ocorrências de um valor dado:

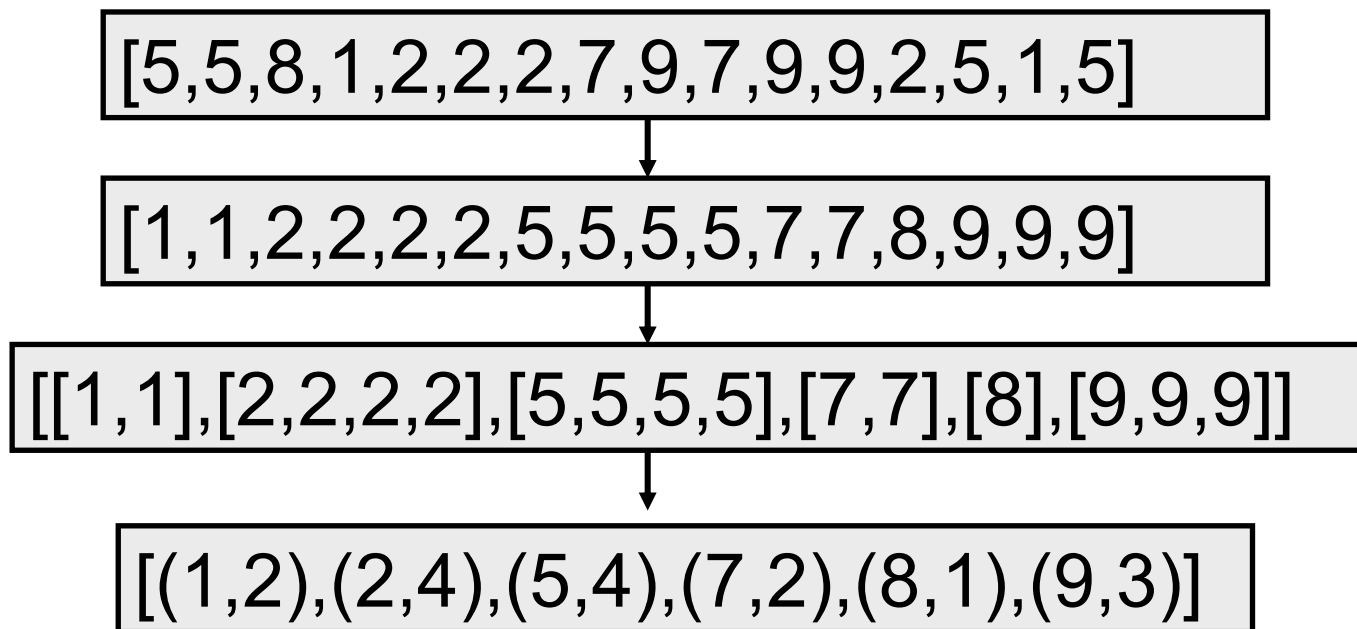
```
count :: Eq a => a -> [a] -> Int  
count x ys = length (filter (==x) ys)
```

- Uma função Haskell vale mais do que mil palavras:

```
countAll :: Eq a => [a] -> [(a, Int)]  
countAll [] = []  
countAll (x:xs) = (x, 1+count x xs): countAll (filter (/=x) xs)
```


Solução por troços

- Estratégia:
 - Ordenar a lista
 - Criar a lista de troços
 - Criar a lista de pares valor – comprimento.



Lista dos troços

- Primeira solução: takeWhile, dropWhile

```
runs :: Eq a => [a] -> [[a]]
runs [] = []
runs (x:xs) = takeWhile (==x) (x:xs) :
              runs (dropWhile (==x) (x:xs))
```

```
countAll_ :: Ord a => [a] -> [(a, Int)]
countAll_ xs =
  map (\x -> (head x, length x)) (runs (qsort xs))
```

Com span, com break

- Que fazem estas funções, span e break?

```
runs_s :: Eq a => [a] -> [[a]]
runs_s [] = []
runs_s (x:xs) = a : runs_s b
  where (a,b) = span (==x) (x:xs)
```

```
runs_b :: Eq a => [a] -> [[a]]
runs_b [] = []
runs_b (x:xs) = a : runs_b b
  where (a,b) = break (/=x) (x:xs)
```

Controlo

- Que funções genéricas do prelúdio aprendemos hoje?
- Como se contam as ocorrências de cada um dos valores presentes numa lista?
- Como se constrói a lista de troços?
- Quantos troços tem a lista repeat 3?

Exercícios

- Programe recursivamente as funções span e break.
- Programe uma função ascendingRuns para construir a lista dos troços ascendentes. Por exemplo: ascendingRuns [1,3,7,6,5,4,8,9,9,1] vale [[1,3,7],[6],[5],[4,8,9,9],[1]].
- Programe uma função para dar o troço mais longo.
- Reprograme a função choose usando a técnica dos troços, com as devidas adaptações.

Na próxima aula

- Voltaremos ao problemas das notas, para produzir a pauta dos exercícios.
- Estas técnicas dos troços vão ser úteis.

Fundamentos da Programação

Vigésima aula:
Questionários e Mergesort

Nesta aula vamos...

- Completar o processamento dos questionários.
- Temos de aprender a ler uma lista de ficheiros, reunindo o resultado operações sobre cada um deles e fazer uma tabela com isso.
- De passagem, estudaremos o mergesort.

Onde estamos?

- Já sabemos processar um ficheiro com resultados dos questionários, por meio da função `readQuizzes'`.
- O resultado é uma lista de pares número de aluno–número de pontos.
- Quando processarmos vários ficheiros, temos de distinguir os resultados de cada, *etiquetando-os* com o número do questionário.

Os nomes dos ficheiros

- Daremos nomes normalizados, sequenciais aos ficheiros: “quiz_01.txt”, “quiz_02.txt”, etc.
- A partir do nome podemos calcular o número do questionário.

```
quizNumber :: FilePath -> Int  
quizNumber xs =  
  read (takeWhile (/= '.') (tail (dropWhile (/= '_') xs)))
```

```
Main> quizNumber "quiz_03.txt"  
3  
Main> quizNumber "aaa_987.txt"  
987
```

Questionários com etiqueta

- O tipo Quiz representa um par número da aluno – número de pontos.

```
type Quiz = (Int, Int)
```

- O tipo QuizLabelled associa o número do questionário ao número de pontos:

```
type QuizLabelled = (Int, (Int, Int))
```

- “Etiquetamos” assim:

```
labelQuiz :: Int -> Quiz -> QuizLabelled  
labelQuiz x (a, b) = (a, (x, b))
```

Afinando o readQuizzes'

- Ao ler, queremos guardar a indicação de que questionário se trata:

```
readQuizzesLabelled ::  
  [Student] -> FilePath -> IO [QuizLabelled]  
readQuizzesLabelled t f =  
  do  
    q <- readQuizzes' t f  
    let x = quizNumber f  
    return (map (labelQuiz x) q)
```

Lendo os ficheiros todos

- Os nomes dos ficheiros vêm numa lista:

```
readAllQuizzesLabelled ::  
  [Student] -> [FilePath] -> IO [QuizLabelled]  
readAllQuizzesLabelled t f =  
  if null f then  
    return ([])  
  else  
    do  
      q <- readQuizzesLabelled t (head f)  
      r <- readAllQuizzesLabelled t (tail f)  
      return (merge q r)
```

Lista de nomes
de ficheiros.

As listas q e r estão orde-
nadas. Ao *fundir*, obtemos
uma lista ordenada.

Fusão de listas

- O algoritmo de fusão de listas constrói uma lista ordenada com os elementos presentes em duas listas ordenadas, numa só passagem:

```
merge :: Ord a => [a] -> [a] -> [a]
```

```
merge [] ys = ys
```

```
merge xs [] = xs
```

```
merge (x:xs) (y:ys)
```

```
  | x <= y = x: merge xs (y:ys)
```

```
  | otherwise = y: merge (x:xs) ys
```

Esta é a variante “com repetição”:
elementos repetidos não são
suprimidos.

```
Main> merge [3,6,7,9,12] [6,8,9,9,11,14,19]
```

```
[3,6,6,7,8,9,9,9,11,12,14,19]
```

```
Main> merge [5,8] []
```

```
[5,8]
```

Ordenação por fusão

- A fusão de listas sugere um algoritmo de ordenação:
 - Parte-se a lista ao meio.
 - Ordena-se (recursivamente) cada uma das metades
 - Fundem-se as duas metades, que já estão ordenadas, obtendo-se uma lista ordenada.
- Como partir uma lista ao meio?

Partir ao meio

- Conhecendo o comprimento:

```
halve :: [a] -> ([a],[a])  
halve xs = splitAt (div (length xs + 1) 2) xs
```

- Sem calcular o comprimento, separando alternadamente os elementos:

```
rip :: [a] -> ([a],[a])  
rip [] = ([],[])  
rip [x] = ([x],[])  
rip (x1:x2:xs) = (x1:a, x2:b)  
    where (a,b) = rip xs
```

```
Main> halve [4,2,9,5,6,3,5]  
([4,2,9,5],[6,3,5])  
Main> rip [4,2,9,5,6,3,5]  
([4,9,6,5],[2,5,3])
```


Mergesort

- Ordenam-se as duas metades da lista e depois fundem-se:

```
msort :: Ord a => [a] -> [a]
msort [] = []
msort [x] = [x]
msort xs = merge (msort a) (msort b)
  where (a, b) = rip xs
```

- Observe:

28-01-2008

Regressando aos questionários

- Função de teste, com quatro ficheiros:

```
testReadAllQuizLabelled :: IO ()
testReadAllQuizLabelled =
  do
    s <- readStudents "alunos_FP0708.txt"
    q <- readAllQuizzesLabelled s quizFilePaths
    putStr
      (unlines (
        map (\(a,(b, c)) ->
          show a ++ " " ++ show b ++ " " ++ show c) q))
```

```
quizFilePaths = ["quiz_01.txt", "quiz_02.txt",
                  "quiz_03.txt", "quiz_04.txt"]
```

```
199 28991 4 1
200 28993 1 1
201 28993 2 1
202 28993 3 1
203 28993 4 1
204 28994 1 1
205 28994 2 1
206 28994 3 1
207 28994 4 1
208 28997 1 1
209 28997 2 1
210 28997 3 1
211 28997 4 1
212 28998 1 1
213 28998 2 1
214 28998 3 1
215 28998 4 1
216 29192 1 0
217 29898 1 1
218 29898 2 1
219 29898 3 1
220 29898 4 1
221 29900 1 0
222 29900 2 1
223 29906 1 1
224 29906 2 1
```

Sumariar as notas

- O passo seguinte é agrupar os pontos de cada aluno, numa lista de pares número do questionário – número de pontos.

```
type QuizSummary = (Int, [(Int, Int)])
```

- O conjunto desses grupos forma a pauta:

```
makeTable :: [QuizLabelled] -> [QuizSummary]
makeTable [] = []
makeTable (x:xs) = (fst x, snd (unzip a)): makeTable b
  where
    (a, b) = span (\x' -> fst x' == fst x) (x:xs)
```

Preparando para o Excel

- Só falta escrever no formato que convém ao Excel, com *tabs* a separar os campos:

```
tabify :: [(Int, Int)] -> String
tabify [] = ""
tabify xs = tabify' xs 1
```

Os campos em falta ficam em branco, entre *tabs*.

```
tabify' :: [(Int, Int)] -> Int -> String
tabify' [] _ = ""
tabify' (x:xs) y
  | fst x == y = chr 9 : (show (snd x) ++ tabify' xs (y+1))
  | otherwise = chr 9 : tabify' (x:xs) (y+1)
```

```
Main> tabify [(2,55),(7,33),(8,66),(12,77)]
"\t\t55\t\t\t\t\t33\t66\t\t\t\t\t77"
Main> tabify [(3,44)]
"\t\t\t44"
```

E pronto!

- Eis a função que escreve a tabela no ficheiro, com o formato desejado:

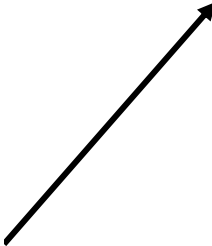
```
testMakeTable :: IO ()
testMakeTable =
  do
    s <- readStudents "alunos_FP0708.txt"
    q <- readAllQuizzesLabelled s quizFilePaths
    let t = makeTable q
    putStr (unlines (map \(a,xs) -> show a ++ tabify xs) t))
```

```
92 31764 1 1 1 1
93 32327 1 1 1 1
94 32627 0 1
95 32925 0
96 32926 1 1 1 1
97 32927 1 1 1 1
98 32929 1
99 32931 1 1 1 1
100 32932 1 1 1
101 32933 1 0 1 1
102 32934 1 1 1 0
103 32936 1 1 1 1
104 32937 0
105 32938 1 1 1 1
106 32939 1 1 1 1
107 33350 1 1 1 1
108 33362 1 1 1 1
109 34725 1 1 1 1
110 34726 1 1 1 0
111 34728 0 1 1
```

No Excel

- Lê-se o ficheiro de texto, indicando que os delimitadores de campo são os *tabs*:

```
92 31764>1>1>1>1CRLF
93 32327>1>1>1>1CRLF
94 32627>→→→0>1CRLF
95 32925>→0CRLF
96 32926>1>1>1>1CRLF
97 32927>1>1>1>1CRLF
98 32929>1CRLF
99 32931>1>1>1>1CRLF
100 32932>→1>1>1CRLF
101 32933>1>0>1>1CRLF
102 32934>1>1>1>0CRLF
103 32936>1>1>1>1CRLF
104 32937>→→→0CRLF
105 32938>1>1>1>1CRLF
106 32939>1>1>1>1CRLF
107 33350>1>1>1>1CRLF
108 33362>1>1>1>1CRLF
109 34725>1>1>1>1CRLF
110 34726>1>1>1>0CRLF
111 34728>0>1>1>1CRLF
```



	A	B	C	D	E	F
88	31758	1	1	1	0	
89	31759	1	1	1	1	
90	31761	1	1	1	1	
91	31762	1	1	1	1	
92	31764	1	1	1	1	
93	32327	1	1	1	1	
94	32627			0	1	
95	32925		0			
96	32926	1	1	1	1	
97	32927	1	1	1	1	
98	32929	1				
99	32931	1	1	1	1	
100	32932		1	1	1	
101	32933	1	0	1	1	
102	32934	1	1	1	0	
103	32936	1	1	1	1	
104	32937				0	
105	32938	1	1	1	1	
106	32939	1	1	1	1	
107	33350	1	1	1	1	
108	33362	1	1	1	1	
109	34725	1	1	1	1	
110	34726	1	1	1	0	
111	34728	0	1	1		
112	34731	1				
113	34732	1	1	1	1	
114	34733	1	1	1	1	
115	34734	1	1	1	1	
116	34736	1	1	1	1	

Controlo

- Quantos algoritmos de ordenação já conhecemos?
- Que função do prelúdio aprendemos hoje?
- Como fizemos para que os campos em falta ficassem em branco, no lugar certo?
- Hoje usámos o unzip para quê?

Exercícios

- Programe a fusão sem repetição.
- Programe a “intersecção” de listas ordenadas, modificando o algoritmo da fusão.
- Programe uma função para partir uma lista em três partes “iguais”, usando o comprimento e sem usar o comprimento.
- Escreva uma função que dado um objecto de tipo QuizSummary e número de questionários, calcule a lista de questionários em falta.

Na próxima aula

- Escreveremos os programas para processar os ficheiros com os resultados do Mooshak.

Fundamentos da Programação

Vigésima primeira aula:
Guiões e mais

Nesta aula vamos...

- Adaptar as funções dos questionários aos ficheiros com resultados dos concursos.
- Na verdade, processar os concursos até é mais simples, pois não é preciso ir buscar o número à lista dos estudantes.
- Como é mais simples, temos tempo para estudar alguns outros assuntos interessantes...

Exemplo de resultados de concurso

- Duas colunas: nome (e grupo) e número de submissões aceites:

```
23 students · ManuelMoreno_32627 · →6CRLF
24 students · Joana_Carolas_32938 · →6CRLF
25 students · marcio_costa_28606 · →6CRLF
26 students · Joao_Valagao_27954 · →6CRLF
27 students · Luis_Silva_34742 · →6CRLF
28 students · Denis_25077 · →6CRLF
29 students · Daniel_Jorge_28991 · →6CRLF
30 students · yanis_yana_34754 · →6CRLF
31 students · Joana_Suzana_Marques_34738 · →6CRLF
32 students · sergio_fernandes_34753 · →6CRLF
33 students · Ana_Leal_34725 · →6CRLF
34 students · Gerson_Fitas_23675 · →6CRLF
35 students · Leonel_Oliveira_25648 · →5CRLF
36 students · Francisco_Machado_28994 · →5CRLF
37 students · Tiago_Susana_30924 · →5CRLF
38 students · Marco_Sousa_29918 · →5CRLF
39 students · Joao_Gomes_31750 · →5CRLF
40 students · Pedro_jesus_25088 · →5CRLF
41 students · Andre_Ramos_30917 · →5CRLF
42 students · Pedro_Vito_29932 · →5CRLF
43 students · Duarte_Marques_31742 · →5CRLF
```

Nem todos os estudantes respeitaram as regras na escolha do nomes de utilizador no Mooshak, mas, para efeitos do processamento das notas, o que conta é o número.

Transformar linha em par

- Reutilizamos o tipo Quiz, que é um par de inteiros número de aluno – número de pontos.
- Ignoramos tudo até ao sublinhado, ficando com duas “palavras”:

```
mooshakResultFromLine :: String -> Quiz
mooshakResultFromLine xs = (a, b)
  where
    z = dropWhile (not . isDigit) xs
    w = words z
    a = read (head w)
    b = read (last w)
```

Composição de funções

- O operador de composição de funções $(.)$ representa uma função de ordem superior.

$$\begin{aligned} (.) &:: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c) \\ f . g &= \lambda x \rightarrow f (g x) \end{aligned}$$

- Logo `not . isDigit` representa a função $\lambda x \rightarrow \text{not } (\text{isDigit } x)$. Neste caso `not` é $\text{Bool} \rightarrow \text{Bool}$ e `isDigit` é $\text{Char} \rightarrow \text{Bool}$.

Soma dos quadrados

- Outro exemplo: função para somar os quadrados dos elementos de uma lista.
- Primeiro uma função para calcular os quadrados:

```
squares :: [Double] -> [Double]
-- squares xs = map (^2) xs
squares = map (^2)
```

- E agora a função pretendida:

```
sumSquares :: [Double] -> Double
-- sumSquares xs = sum (squares xs)
-- sumSquares = sum . squares
sumSquares = sum . map (^2)
```

Na prática, normalmente dispensamos estas duas funções, escrevendo simplesmente `map (^2)` e `sum . map (^2)`.

Exemplos de utilização

- Calcular a soma dos quadrados para uma lista de listas:

```
Main> map (sum.map(^2)) [[5,2,8,6],[],[4],[2,7]]  
[129,0,16,53]
```

- Eliminar as listas vazias de uma lista de listas:

```
Main> filter (not.null) [[4],[5,6,8],[],[],[5]]  
[[4],[5,6,8],[5]]
```

- Os comprimentos das palavras:

```
Main> map length (words "aaa bbbbbb ff")  
[3,6,2]  
Main> (map length . words) "aaa bbbbbb ff"  
[3,6,2]
```

Como se programa o words?

- Talvez assim:

```
words_ :: String -> [String]
words_ [] = []
words_ xs = words_ (dropWhile isSpace xs)

words_ ' ' :: String -> [String]
words_ ' ' [] = []
words_ ' ' xs = a : words_ b
  where (a, b) = break isSpace xs
```

```
Main> words "castelo branco"
["castelo","branco"]
Main> words "    aaa    bbb    ccc    "
["aaa","bbb","ccc"]
Main> words " "
[]
```

Generalizando

- A função `tokens` cria uma lista de listas a partir de uma lista, usando como *delimitadores* os elementos que satisfazem um predicado dado:

```
tokens :: (a -> Bool) -> [a] -> [[a]]
tokens _ [] = []
tokens p xs = tokens' p (dropWhile p xs)

tokens' :: (a -> Bool) -> [a] -> [[a]]
tokens' _ [] = []
tokens' p xs = a : tokens p b
  where (a, b) = break p xs
```

Generalizando

- Com vários tipos de listas:

```
Main> tokens (== ',') "lisboa,coimbra,faro"
["lisboa","coimbra","faro"]
Main> tokens (<0) [8,9,-1,6,8,2,1,-4,-5,7]
[[8,9],[6,8,2,1],[7]]
Main> tokens (not . isAlphaNum) "quiz_01.txt"
["quiz","01","txt"]
Main> tokens (== 0) [0,0,0]
[]
Main> tokens (== 0) [0,0,5,0,6,0,0]
[[5],[6]]
```

- Aliás, a função words é um caso particular:

```
words_t :: String -> [String]
-- words_t xs = tokens isSpace xs
words_t = tokens isSpace
```

Como se programa o read?

- Para inteiros:

```
readIntegral :: Integral a => String -> a
readIntegral "" = error "integralFromString: empty string"
readIntegral xs = readIntegral' 0 xs

readIntegral' :: Integral a => a -> String -> a
readIntegral' v [] = v
readIntegral' v (x:xs)
  | isDigit x = readIntegral' (10*v + fromIntegral(digitToInt x)) xs
  | otherwise =
    error ("readIntegral: not a digit: " ++ "\"" ++ [x] ++ "\"")
```

Por exemplo, “528” dá $((10 * 0 + 5) * 10 + 2) * 10 + 8$.

Ou então usando foldl

```
readIntegral_ :: Integral a => String -> a
```

```
readIntegral_ = foldl (\v x -> 10 * v + fromIntegral (digitToInt x)) 0
```

Exemplos:

```
Main> readIntegral "747723"
```

```
747723
```

```
Main> readIntegral "4"
```

```
4
```

```
Main> readIntegral ""
```

```
Program error: readIntegral: empty string
```

```
Main> readIntegral_ "111222333444555"
```

```
111222333444555
```

```
Main> readIntegral_ ""
```

```
0
```

```
Main> readIntegral_ "27o"
```

```
Program error: Char.digitToInt: not a digit 'o'
```

```
Main> readIntegral_ "aa"
```

```
110
```

```
Main> readIntegral "aa"
```

Regressando ao problema

- Prosseguimos analogamente ao caso dos questionários:

```
readMooshak :: FilePath -> IO [Quiz]
readMooshak f =
  do
    s <- readFile f
    return (map mooshakResultFromLine (lines s))

readMooshakLabelled :: FilePath -> IO [QuizLabelled]
readMooshakLabelled f =
  do
    q <- readMooshak f
    let x = quizNumber f
    return (qsort (map (labelQuiz x) q))
```

Função da
página 4.

Construindo a tabela

- Lemos os ficheiros todos e tabelamos:

```
readAllMooshakLabelled :: [FilePath] -> IO [QuizLabelled]
readAllMooshakLabelled f =
  if null f then
    return ([])
  else
    do
      q <- readMooshakLabelled (head f)
      r <- readAllMooshakLabelled (tail f)
      return (merge q r)

testMooshakTable :: [FilePath] -> IO ()
testMooshakTable fs =
  do
    q <- readAllMooshakLabelled fs
    let t = makeTable q
    putStr (unlines (map \(a,xs) -> show a ++ tabify xs) t))
```


Construindo a tabela

- Com os três guiões e os problemas:

```
scriptFilePaths = ["g_1.txt", "g_2.txt", "g_3.txt", "p_4.txt"]
```

```
testScripts :: IO ()
```

```
testScripts = testMooshakTable scriptFilePaths
```

```
94 32925 5
95 32926 4 9 4 2
96 32929 4 2
97 32931 5 9 6 2
98 32932 5 9 3 2
99 32933 5 9 5 2
100 32934 2 2 2
101 32935 1
102 32937 5 8 5 2
103 32938 5 9 6 2
104 32939 5 9 6 2
105 33350 5 9 2 2
106 33362 4
107 34725 5 9 6 2
108 34726 5 9
109 34727 9
110 34729 2 3
111 34730 5 9 4 2
112 34731 4 9
113 34732 3 9 1
114 34733 4 9
```

```
94 32925>5<CRLF
95 32926>4>9>4>2<CRLF
96 32929>4>2<CRLF
97 32931>5>9>6>2<CRLF
98 32932>5>9>3>2<CRLF
99 32933>5>9>5>2<CRLF
100 32934>2>2>→2<CRLF
101 32935>→1<CRLF
102 32937>5>8>5>2<CRLF
103 32938>5>9>6>2<CRLF
104 32939>5>9>6>2<CRLF
105 33350>5>9>2>2<CRLF
106 33362>4<CRLF
107 34725>5>9>6>2<CRLF
108 34726>5>9<CRLF
109 34727>→9<CRLF
110 34729>2>3<CRLF
111 34730>5>9>4>2<CRLF
112 34731>4>9<CRLF
113 34732>3>9>→1<CRLF
114 34733>4>9<CRLF
```

Mostrando todos
os caracteres

Controlo

- Qual é a assinatura da composição de funções?
- Que exemplo de fold vimos hoje?
- Como se converte de String para inteiro?
- Se em vez de usar tabs como delimitadores usássemos vírgulas, o nosso programa ficava mais simples ou menos simples?
- Tratar os ficheiros com os resultados dos concursos é mais simples do que tratar os dos questionários. Porquê?
- Como se representa o carácter plica numa cadeia de caracteres?

Exercícios

- Modifique a função `readIntegral_` de maneira a dar erro nos casos inválidos.
- Programe o `unwords`.
- Programe o `untokens`, usando como separador um valor dado.
- Programe um função `roman`, que dado um número calcula uma cadeia que é a representação desse numero em notação romana.
- Programe uma função `readRoman`, que faz o inverso da anterior.

Na próxima aula

- Inspirados pela aula de hoje, veremos as questões da representação numérica nos computadores.

Fundamentos da Programação

Vigésima segunda aula:
Representação numérica

Nesta aula vamos...

- Estudar algumas questões relativas à representação dos números nos computadores.
- Hoje veremos a representação decimal.
- Amanhã, a representação binária.

Representação decimal

- Os números podem ter várias representações, em geral.
- Nos programas, normalmente representamos os números inteiros usando a notação decimal.
- “Internamente”, os números são representados binariamente.
- Estudaremos primeiro a notação decimal por meio de uma série de pequenos exercícios.

Problema fundamental da programação 😊

- Saber se um número é uma capicua!
- Uma capicua é um número que se lê igualmente do princípio para o fim e do fim para o princípio.
- Ser capicua não é uma propriedade do número apenas, como ser primo, mas do número e da sua representação decimal.

Capicuas

- Um número é uma capicua se a sua representação decimal for uma cadeia simétrica.

“Capicua” é uma palavra de origem catalã. “Palíndromo” é uma palavra de origem grega. Têm o mesmo significado.

```
isSymmetric :: String -> Bool  
isSymmetric xs = xs == reverse xs
```

```
isPalindrome :: Integral a => a -> Bool  
isPalindrome = isSymmetric . show
```

```
Main> isPalindrome 234  
False  
Main> isPalindrome 23432  
True  
Main> isPalindrome 2  
True  
Main> isPalindrome 111222333444444333222111  
True  
Main> isPalindrome 00100  
False  
Main> isPalindrome 1001  
True  
...
```

E se não houvesse o show?

- Se não houvesse, programávamo-lo nós:

```
showIntegral_ :: Integral a => a -> String
```

```
showIntegral_ x
```

```
  | x < 10 = [integralToDigit x]
```

```
  | otherwise =
```

```
    showIntegral_ (div x 10) ++ [integralToDigit (mod x 10)]
```

OK, mas não é boa ideia ter um ++ dentro da parte recursiva, pois torna o algoritmo quadrático.

```
integralToDigit :: Integral a => a -> Char
```

```
integralToDigit x = chr (fromIntegral x + ord '0')
```

Esta é necessária porque o intToDigit só dá para argumentos Int. Não é “general enough”...

Nova tentativa

- Com parâmetro de acumulação, onde vamos acumulando a parte final da representação, da direita para a esquerda:

```
showIntegral :: Integral a => a -> String
showIntegral 0 = "0"
showIntegral x = showIntegral' "" x
```

```
showIntegral 5839 =
showIntegral' "" 5839 =
showIntegral' "9" 583 =
showIntegral' "39" 58 =
showIntegral' "839" 5 =
showIntegral' "5839" 0 =
"5839"
```

```
showIntegral' :: Integral a => String -> a -> String
showIntegral' s 0 = s
showIntegral' s x =
    showIntegral' (integralToDigit (mod x 10) : s) (div x 10)
```

Quantos algoritmos tem um número

- Versão preguiçosa:

```
countDigits_ :: Integral a => a -> Int  
countDigits_ = length . show
```

- Versão aritmética:

```
countDigits :: Integral a => a -> Int  
countDigits x  
  | x < 10 = 1  
  | otherwise = 1 + countDigits (div x 10)
```

A versão aritmética
é preferível.

A soma dos algarismos

- Versão preguiçosa:

```
sumDigits_ :: Integral a => a -> a
-- sumDigits_ x = sum (map digitToIntegral (show x))
sumDigits_ = sum . map digitToIntegral . show
```

```
digitToIntegral :: Integral a => Char -> a
digitToIntegral x = fromIntegral (ord x - ord '0')
```

- Versão aritmética:

```
sumDigits :: Integral a => a -> a
sumDigits x
  | x < 10 = x
  | otherwise = x `mod` 10 + sumDigits (x `div` 10)
```

A versão aritmética
é preferível.

Invertendo aritmeticamente

- Versão preguiçosa:

```
reverseIntegral_ :: Integral a => a -> a
-- reverseIntegral_ x = fromIntegral (read (reverse (show x)))
reverseIntegral_ = fromIntegral . read . reverse . show
```

- Versão aritmética:

```
reverseIntegral :: Integral a => a -> a
reverseIntegral x = reverseIntegral' 0 x
```

```
reverseIntegral' :: Integral a => a -> a -> a
reverseIntegral' v 0 = v
reverseIntegral' v x =
    reverseIntegral' (10 * v + x `mod` 10) (x `div` 10)
```

```
reverseIntegral 5839 =
reverseIntegral' 0 5839 =
reverseIntegral' 9 583 =
reverseIntegral' 93 58 =
reverseIntegral' 938 5 =
reverseIntegral' 9385 0 =
9385
```

Técnica do parâmetro
de acumulação.

A verdadeira capicua

- A verdadeira capicua programa-se aritmeticamente!

```
isPalindrome :: Integral a => a -> Bool  
isPalindrome x = x == reverseIntegral x
```

- As capicuas com três algarismos:

```
Main> filter isPalindrome [100..999]  
[101,111,121,131,141,151,161,171,181,191,202  
,212,222,232,242,252,262,272,282,292,303,313  
,323,333,343,353,363,373,383,393,404,414,424  
,434,444,454,464,474,484,494,505,515,525,535  
,545,555,565,575,585,595,606,616,626,636,646  
,656,666,676,686,696,707,717,727,737,747,757  
,767,777,787,797,808,818,848,858,868,878,888  
,898,909,919,929,939,949,959,969,979,989,999  
]
```

Leveza

- Dizemos que x é *mais leve* do que y se a soma dos algarismos de x for menor que a soma dos algarismos de y ou, sendo iguais as somas, se x for menor ou igual a y :

```
lighterThan :: Integral a => a -> a -> Bool
```

```
lighterThan x y
```

```
  | a == b = x <= y
```

```
  | otherwise = a <= b
```

```
  where
```

```
    a = sumDigits x
```

```
    b = sumDigits y
```

```
Main> lighterThan 45 39
```

```
True
```

```
Main> lighterThan 66 91
```

```
False
```

```
Main> lighterThan 38 87
```

```
True
```

```
Main> lighterThan 100 31
```

```
True
```

```
Main> lighterThan 37 28
```

```
False
```

```
Main> lighterThan 87 96
```

```
True
```


Ordenação por leveza

- Recordemos o quicksort:

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

- Para ordenar por leveza, deve bastar mudar os filtros, de `<= x` por `lighterThan x` e `>x` por `not . lighterThan x`:

```
qsort_lighter :: Integral a => [a] -> [a]
```

```
qsort_lighter [] = []
```

```
qsort_lighter (x : xs) =
```

```
    qsort_lighter (filter (lighterThan x) xs) ++
```

```
    [x] ++
```

```
    qsort_lighter (filter (not . lighterThan x) xs)
```

Hmm, sai ao contrário...

```
Main> qsort_lighter [65,45,99,39,4,110,3,2]
[99,39,65,45,4,3,110,2]
Main> qsort_lighter [1017,88,9,15,39,6]
[88,39,1017,9,15,6]
```

Analizando os filtros

- De facto, a secção ($\leq x$) representa o predicado $\lambda x' \rightarrow x' \leq x$.
- A expressão `lighterThan x` representa o predicado $\lambda x' \rightarrow \text{lighterThan } x \ x'$.
- O que queremos é $\lambda x' \rightarrow \text{lighterThan } x' \ x$:

```
qsort_lighter :: Integral a => [a] -> [a]
```

```
qsort_lighter [] = []
```

```
qsort_lighter (x : xs) =
```

```
    qsort_lighter (filter p xs) ++
```

```
    [x] ++
```

```
    qsort_lighter (filter (not . p) xs)
```

```
    where p =  $\lambda x' \rightarrow \text{lighterThan } x' \ x$ 
```

```
Main> qsort_lighter [65,45,99,39,4,110,3,2]  
[2,110,3,4,45,65,39,99]
```

```
Main> qsort_lighter [1017,88,9,15,39,6]  
[6,15,9,1017,39,88]
```

Agora sim!

Observação final

- Podemos evitar a expressão lambda, usando o flip.
- Com efeito flip (\leq) é $\lambda x y \rightarrow y \leq x$
- E flip lighter é $\lambda x y \rightarrow \text{lighter } y \ x$
- Logo, (flip lighter) x é $\lambda y \rightarrow \text{lighter } y \ x$.
- Assim, podemos reescrever o where:

```
...  
qsort_lighter (x : xs) =  
  ...  
  where p = (flip lighterThan) x
```

Controlo

- O que é uma capicua?
- O que é um palíndromo?
- Como se programa o show?
- Como se definem, tipicamente, critérios de ordenação compostos?
- Que cuidados devemos ter ao usar o quicksort com critérios compostos?
- Quais foram os exemplos de funções compostas de hoje?

Exercícios

- Programe uma função para calcular o número de algarismos '1' presentes na representação decimal de um número dado. Generalize para contar um algarismo qualquer, dado.
- Programe uma função para verificar se um número é um “super-primo”. Um super-primo é um número que tal que, apagando qualquer número de algarismos à direita, o número obtido também é primo.
- Programe uma função que dado um número x calcula a lista dos números que se obtêm apagando cada um dos algarismos de x .
- Programe uma função para o valor do algarismo mais significativo de um número.
- Programe uma função para verificar se os algarismos de um número vêm por ordem crescente.
- Programe uma função para ordenar uma lista de números pelo último algarismo, desempatando pelo valor do número. Por exemplo [15,223,71,111] dá [71,111,223,15].

Na próxima aula

- Estudaremos a representação binária.

Fundamentos da Programação

Vigésima terceira aula:
Números binários

Nesta aula vamos...

- Reaprender as conversões de e para a base 2, programando-as.
- Implementar os algoritmos de adição e multiplicação na base 2.
- São exercícios muito instrutivos.

readBinary

- Só pode ser análoga à função `readIntegral`, substituindo 10 por 2:

```
readIntegral :: Integral a => String -> a
readIntegral = foldl (\v x -> 10 * v + fromIntegral (digitToInt x)) 0
```

```
readBinary :: Integral a => Binary -> a
readBinary =
  foldl (\v x -> 2 * v + fromIntegral (digitToInt x)) 0
```

```
Main> readBinary "110011"
51
Main> readBinary "1"
1
Main> readBinary "100000"
32
Main> readBinary ("1" ++ replicate 100 '0')
1267650600228229401496703205376
Main> 2^100
1267650600228229401496703205376
Main> readBinary (replicate 100 '1')
1267650600228229401496703205375
```

Usamos o tipo `Binary` para as cadeias formadas só por '0' e '1':

```
type Binary = String
```

A lista das potências

- É uma lista infinita:

```
powers :: Num a => a -> [a]
powers x = iterate (*x) 1
```

```
Main> take 11 (powers 2)
[1,2,4,8,16,32,64,128,256,512,1024]
Main> take 10 (powers 0.5)
[1.0,0.5,0.25,0.125,0.0625,0.03125,0.015625,
0.0078125,0.00390625,0.001953125]
Main> take 4 (powers 10)
[1,10,100,1000]
Main> take 5 (powers (exp 1))
[1.0,2.71828182845905,7.38905609893065,20.08
55369231877,54.5981500331442]
```

iterate

- A função `iterate` cria uma lista infinita aplicando sucessivamente uma função dada, a partir de um valor inicial:

```
iterate_ :: (a -> a) -> a -> [a]
iterate_ f x = x : iterate_ f (f x)
```

```
Main> take 12 (iterate_ (+1) 0)
[0,1,2,3,4,5,6,7,8,9,10,11]
Main> take 11 (iterate_ (*2) 1)
[1,2,4,8,16,32,64,128,256,512,1024]
Main> take 10 (iterate_ ('a':) [])
["", "a", "aa", "aaa", "aaaa", "aaaaa", "aaaaaa", "aaaaaaa", "aaaaaaaa", "aaaaaaaaa"]
Main> take 8 (iterate_ (^2) 2)
[2,4,16,256,65536,4294967296,18446744073709551616,340282366920938463463374607431768211456]
Main> take 8 (iterate_ (++"bla") "bla")
["bla", "blabla", "blablabla", "blablablabla", "blablablablabla", "blablablablablabla", "blablablablablablabla", "blablablablablablablabla"]
```

Vontade de complicar ☹

- Alternativamente, para a leitura binária, podíamos ter construído a lista das potências de 2, multiplicando-as com os valores dos algarismos binários respectivos por ordem inversa, somando tudo no fim:

```
readBinaryAlt :: Integral a => String -> a
readBinaryAlt xs =
    sum (zipWith (*) (map digitToIntegral (reverse xs)) (powers 2))
```

A função zipWith

- Generaliza a função zip, especificando a função binária que se aplica a cada par de elementos, um de cada lista:

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

- A função zip é um caso particular dessa:

```
zip_ :: [a] -> [b] -> [(a,b)]
-- zip_ = zipWith (\x y -> (x,y))
zip_ = zipWith (,)
```

showBinary

- É análoga a showIntegral, claro:

```
showIntegral :: Integral a => a -> String
showIntegral 0 = "0"
showIntegral x = showIntegral' "" x
```

```
showIntegral' :: Integral a => String -> a -> String
showIntegral' s 0 = s
showIntegral' s x = showIntegral' (integralToDigit (mod x 10) : s) (div x 10)
```

```
showBinary :: Integral a => a -> Binary
showBinary 0 = "0"
showBinary x = showBinary' "" x
```

```
showBinary' :: Integral a => Binary -> a -> Binary
showBinary' s 0 = s
showBinary' s x = showBinary' (integralToDigit (mod x 2) : s) (div x 2)
```

```
Main> showBinary 16
"10000"
Main> showBinary 80
"1010000"
Main> showBinary 155
"10011011"
Main> showBinary 7
"111"
Main> showBinary 1
"1"
```

Números binários, de cabeça

- Exemplo: $6 = 4 + 2$, logo $6 = 110_2$.
- Exemplo: $13 = 8 + 4 + 1$, logo $13 = 1101_2$.
- Exemplo: $25 = 16 + 8 + 1$, logo $25 = 11001_2$.
- Exemplo: $48 = 32 + 16$, logo $48 = 110000_2$.
- Exemplo: $87 = 64 + 16 + 4 + 2 + 1$, logo $87 = 1010111_2$.
- Exemplo: $172 = 128 + 32 + 8 + 4$, logo $172 = 10101100_2$.

Aritmética binária

- O tipo Binary não é bom para fazer contas, porque o algarismo menos significativo, por onde começamos as operações, é o último da cadeia
- Para a aritmética binária, representamos os números por listas de booleanos, com o dígito binário menos significativo à cabeça:

```
type Bits = [Bool]
```


Conversão entre Binary e Bits

- Observe:

```
showBits :: Bits -> Binary
-- showBits xs = reverse (map (intToDigit.fromEnum) xs)
showBits = reverse . map (intToDigit.fromEnum)
```

```
readBits :: Binary -> Bits
readBits = reverse . map (== '1')
```

```
Main> showBinary 12
"1100"
Main> (readBits . showBinary) 12
[False,False,True,True]
Main> showBits [True, True, False, False, True]
"10011"
Main> (readBinary . showBits) [True, True,
False, False, True]
19
```

Sucessor binário

- Para somar 1 a um número escrito na base 2, começa-se pelo fim, mudando todos os '1' para '0' até aparecer um '0', o qual passa para '1'. Se chegarmos ao início do número, acrescentamos um '1'.
- Por exemplo, $11001_2 + 1$ dá 11010_2 ; $100111_2 + 1$ dá 101000_2 ; $111_2 + 1$ dá 1000_2 .
- Programamos primeiro em termos do tipo Bits e depois adaptamos para Binary.

Sucessor com Bits

- É um exercício de listas:

```
succBits :: Bits -> Bits
succBits [] = [True]
succBits (False : xs) = True : xs
succBits (True : xs) = False : succBits xs
```

```
succBinary :: Binary -> Binary
succBinary = showBits . succBits . readBits
```

```
Main> succBinary "11001"
"11010"
Main> succBinary "100111"
"101000"
Main> succBinary "111"
"1000"
Main> succBinary "1111111111111111"
"10000000000000000000"
Main> succBinary "11000"
"11001"
```

Menor binário

- Muito interessante:

```
lessThanBits :: Bits -> Bits -> Bool
lessThanBits [] _ = True
lessThanBits (x:xs) [] = False
lessThanBits (x:xs) (y:ys)
    | xs == ys = x <= y
    | otherwise = lessThanBits xs ys
```

```
lessThanBinary :: Binary -> Binary -> Bool
lessThanBinary x y = lessThanBits (readBits x) (readBits y)
```

```
Main> lessThanBinary "11100" "11111110"
True
Main> lessThanBinary "11100" "1111"
False
Main> lessThanBinary "11100" "11110"
True
Main> lessThanBinary "11100" "11000"
False
Main> lessThanBinary "11111" "1111"
False
Main> lessThanBinary "111" "1111"
True
```

Adição com Bits

- Somamos da esquerda para a direita, tendo em atenção o transporte:

```
addBits :: Bits -> Bits -> Bits
addBits xs ys = addBits' False xs ys
```

```
addBits' :: Bool -> Bits -> Bits -> Bits
```

```
addBits' False xs [] = xs
```

```
addBits' False [] ys = ys
```

```
addBits' True xs [] = succBits xs
```

```
addBits' True [] ys = succBits ys
```

```
addBits' False (False:xs) (False:ys) = False: addBits' False xs ys
```

```
addBits' False (False:xs) (True:ys) = True: addBits' False xs ys
```

```
addBits' False (True:xs) (False:ys) = True: addBits' False xs ys
```

```
addBits' False (True:xs) (True:ys) = False: addBits' True xs ys
```

```
addBits' True (False:xs) (False:ys) = True: addBits' False xs ys
```

```
addBits' True (False:xs) (True:ys) = False: addBits' True xs ys
```

```
addBits' True (True:xs) (False:ys) = False: addBits' True xs ys
```

```
addBits' True (True:xs) (True:ys) = True: addBits' True xs ys
```

Análise por casos, fastidiosa. Mas podemos simplificar.

Simplificando addBits'

- Reunimos os casos em que o transporte é False e aqueles em que é True:

```
addBits' :: Bool -> Bits -> Bits -> Bits
addBits' False xs [] = xs
addBits' False [] ys = ys
addBits' True xs [] = succBits xs
addBits' True [] ys = succBits ys
addBits' False (x:xs) (y:ys) = (x/=y) : addBits' (x && y) xs ys
addBits' True (x:xs) (y:ys) = (x==y) : addBits' (x || y) xs ys
```

Adição binária

- Agora é simples:

```
addBinary :: Binary -> Binary -> Binary
```

```
addBinary x y = showBits (addBits (readBits x) (readBits y))
```

```
Main> showBinary 815
"1100101111"
Main> showBinary 546
"1000100010"
Main> addBinary "1100101111" "1000100010"
"10101010001"
Main> readBinary "10101010001"
1361
Main> 815+546
1361
Main> addBinary "111" "10"
"1001"
Main> addBinary "111000" "101"
"111101"
Main> addBinary "101010" "110"
"110000"
```

Multiplicação

- Usamos a formulação recursiva do algoritmo da escola primária:

$\text{mult10} :: \text{Integral } a \Rightarrow a \rightarrow a \rightarrow a$

$\text{mult10 } x \ 0 = 0$

$\text{mult10 } x \ y = \text{mult10 } (10 * x) (\text{div } y \ 10) + x * (\text{mod } y \ 10)$

Por hipótese, sabemos multiplicar por 10: basta acrescentar um '0'; sabemos dividir por 10: basta apagar o algarismo mais à direita; e sabemos multiplicar por um número entre 1 e 9.

$\text{mult10 } 2387 \ 572 =$

$\text{mult10 } 23870 \ 57 + 2387 * 2 =$

$\text{mult10 } 238700 \ 5 + 23870 * 7 + 2387 * 2 =$

$\text{mult10 } 2387000 \ 0 + 238700 * 5 + 23870 * 7 + 2387 * 2 =$

$0 + 238700 * 5 + 23870 * 7 + 2387 * 2$

Formulação binária

- É a mesma coisa:

```
mult2 :: Integral a => a -> a -> a
mult2 x 0 = 0
mult2 x y = mult2 (2 * x) (div y 2) + x * (mod y 2)
```

- Ou então, separando o caso em que y é par e o caso em que não é:

```
mult2' :: Integral a => a -> a -> a
mult2' x 0 = 0
mult2' x y
  | even y = mult2' (2 * x) (div y 2)
  | otherwise = mult2' (2 * x) (div y 2) + x
```

Multiplicação de bits

- Adaptamos a função `mult2'`:

```
multBits :: Bits -> Bits -> Bits
multBits xs [] = [False]
multBits xs (False:ys) = multBits (False:xs) ys
multBits xs (True:ys) = addBits (multBits (False:xs) ys) xs
```

```
multBinary :: Binary -> Binary -> Binary
multBinary xs ys =
    showBits (multBits (readBits xs) (readBits ys))
```

```
Main> multBinary "111110001" "100101010"
"100100001010001010"
Main> readBinary "111110001"
497
Main> readBinary "100101010"
298
Main> readBinary "100100001010001010"
148106
Main> 497*298
148106
```

Operadores

- Podemos definir operadores para tornar as coisas mais interessantes:

```
(+|+) :: Binary -> Binary -> Binary  
(+|+) = addBinary
```

```
(*|*) :: Binary -> Binary -> Binary  
(*|*) = multBinary
```

```
Main> "110011" +|+ "111"  
"111010"  
Main> "1111" *|* "1010"  
"10010110"
```

Controlo

- Que funções do prelúdio apareceram hoje?
- Como se adiciona na base 2? E como se multiplica?
- Como se comparam números escritos na base 2 para ver qual é o menor?
- E como se compara a igualdade? Usámos o operador `==`, mas podíamos ter programado isso nós próprios.

Exercícios

- Generalize as funções `readIntegral` e `readBinary` de maneira a funcionar com qualquer base.
- Idem para `showIntegral` e `showBinary`.
- Programe as funções `predBits` e `predBinary`, para calcular o predecessor.
- Programe a subtracção binária, na hipótese de o aditivo ser maior ou igual ao subtrativo.

Na próxima aula

- Desvendaremos o segredo do Haskell.

Fundamentos da Programação

Vigésima quarta aula:
Segredos do Haskell

Nesta aula vamos...

- Revelar os três segredos do Haskell.
- E despedir-nos de Fundamentos da Programação ☹.

Primeiro segredo

- As funções têm um argumento e um resultado.
- Isto é, só um argumento e só um resultado!
- Só um resultado, já nós sabíamos.
- Mas não estamos nós fartos de usar funções com múltiplos argumentos?

Funções com múltiplos argumentos

- Em rigor, uma assinatura $f :: a \rightarrow b \rightarrow c$ deve interpretar-se $f :: a \rightarrow (b \rightarrow c)$.
- Isto é, em rigor, $f :: a \rightarrow b \rightarrow c$ significa que f é uma função com um argumento de tipo a e resultado de tipo $b \rightarrow c$.
- Ora $b \rightarrow c$ é o tipo das funções de b para c .
- Ou seja, neste caso, em rigor, a expressão $f\ x$, para um dado x , representa uma função de tipo $b \rightarrow c$, e não uma função com dois argumentos.

Exemplos

`elem_ :: Int -> [Int] -> Bool`

- Podemos dizer, imaginando os parêntesis, que a função `elem_` tem um argumento de tipo `Int` e um resultado que é uma função `[Int] -> Bool`.
- Realmente `elem x`, para um `x` dado, é a função que aplicada a uma lista dá `True` se `x` existir na lista e `False` se não.

```
Main> (elem 4) [1, 3..10]
False
Main> (elem 'i') "albufeira"
True
```

Vogais

- O seguinte predicado caracteriza as vogais minúsculas:
- Todas as variantes em comentário são válidas:

```
isVowel :: Char -> Bool
-- isVowel x = elem x "aeiou"
-- isVowel x = flip elem "aeiou" x
-- isVowel x = ((flip elem) "aeiou") x
isVowel = flip elem "aeiou"
```

```
Main> isVowel 'a'
True
Main> isVowel 't'
False
```

Caso com três argumentos

- Uma função $g :: a \rightarrow b \rightarrow c \rightarrow d$, é, em rigor, uma função de $a \rightarrow (b \rightarrow (c \rightarrow d))$.
- Escrever $f\ x\ y\ z$ é o mesmo que escrever $((f\ x)\ y)\ z$ ou $(f\ x)\ x\ y$.
- Exemplo: distância no movimento uniformemente acelerado:

```
dist :: Double -> (Double -> (Double -> Double))  
dist a v0 t = 0.5 * a * t^2 + v0 * t
```

```
Main> dist 2 10 5
```

```
75.0
```

```
Main> (dist 2 10) 5
```

```
75.0
```

```
Main> (dist 2) 10 5
```

```
75.0
```

```
Main> flip (dist 2) 5 10 7
```

```
75.0
```

Outro exemplo

```
replaceThese :: Eq a => [a] -> a -> ([a] -> [a])  
--replaceThese xs y zs =  
--    map (\x -> if elem x xs then y else x) zs  
replaceThese xs y = map (\x -> if elem x xs then y else x)
```

- Portanto `replaceThese xs y`, para `xs` e `y` dados, é uma função que aplicada a uma lista substitui todos as ocorrências de um valor presente em `xs` por `y`. Vendo assim, o argumento é a lista.

```
Main> (replaceThese ['0'..'9'] '*') "23 de Janeiro de 2008"  
"23 de Janeiro de 2008"  
Main> (replaceThese [8, 9] 10) [16, 6, 8, 10, 9, 15]  
[16,6,10,10,10,15]
```

E ainda

```
replaceThese :: Eq a => [a] -> (a -> [a] -> [a])  
replaceThese xs y zs = ...
```

- Também é verdade `replaceThese xs`, para `xs` dado, é uma função com dois argumentos, um de tipo `a` e outro de tipo `[a]`, a qual quando aplicada a um dado valor e a uma dada lista substitui por esse valor todas as ocorrências na lista de algum dos elementos de `xs`.

```
Main> (replaceThese "ckqsz") 'z' "cassiopeia"  
"zazziopeia"  
Main> (replaceThese [0..9]) 0 [6, 17, 9, 12, 15]  
[0,17,0,12,15]
```

Portanto...

- Em rigor, em Haskell, todas as funções têm um só argumento e um só resultado.
- O resultado pode ser uma função. É o que acontece quando “parece” que a função tem múltiplos argumentos.
- Quando escrevemos $f :: a \rightarrow b \rightarrow c$ estamos a dizer, em rigor, que f é uma função com um argumento de tipo a e resultado de tipo $b \rightarrow c$.
- Na prática, quase sempre continuaremos a escrever $f\ x\ y$ com o sentido habitual.
- Ocasionalmente, convém perceber claramente que, com esta função f , $f\ x$, para um x dado, é uma função de tipo $b \rightarrow c$.

Disfarçando

- Se não tivéssemos mentido, e não quiséssemos confusões com funções que dão funções, podíamos ter disfarçado usando pares como argumentos. Por exemplo:

```
modulus :: (Double, Double) -> Double
modulus (x, y) = sqrt (x^2 + y^2)
```

```
Main> modulus 3 4
ERROR - Type error in application
*** Expression      : modulus 3 4
*** Term            : modulus
*** Type            : (Double,Double) -> Double
*** Does not match : a -> b -> c
```

```
Main> modulus (3 4)
ERROR - Cannot infer instance
*** Instance       : Num (a -> (Double,Double))
*** Expression     : modulus (fromInt 3 4)
```

```
Main> modulus (3,4)
5.0
```

curry

- A função `curry` transforma uma função de pares numa função “normal”:

Haskell Curry, claro.

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

```
Main> curry modulus 3 4
5.0
```

```
Main> curry_ modulus 3 4
5.0
```

- As funções com múltiplos argumentos que afinal são funções com um argumento e para as quais o resultado é uma função chamam-se funções “curried” (em português, “curriadas”).

uncurry

- A função uncurry transforma uma função *curried* numa função com um argumento que é um par:

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

```
Main> elem 4 [1..8]
True
Main> uncurry elem (4, [1..8])
True
```

```
Main> addBinary "11001" "1000"
"100001"
Main> uncurry addBinary ("11001","1000")
"100001"
```

Problema: quantos pontos tem o meu clube?

- Dados os resultados, numa lista de pares, calcular os pontos:

```
victories :: [(Int, Int)] -> [(Int, Int)]
```

```
-- victories xs = filter (uncurry (>)) xs
```

```
victories = filter (uncurry (>))
```

```
ties :: [(Int, Int)] -> [(Int, Int)]
```

```
-- victories xs = filter (uncurry (>)) xs
```

```
ties = filter (uncurry (==))
```

```
points :: [(Int, Int)] -> Int
```

```
points xs = 3 * length (victories xs) + length (ties xs)
```

```
Main> points [(2,0),(1,1),(3,1),(0,1)]
```

```
7
```

```
Main> points []
```

```
0
```

```
Main> points [(2,2),(0,0),(1,2)]
```

```
2
```

```
Main> uncurry (<)(8,3)
```

```
False
```

```
Main> uncurry (==)(4,2+2)
```

```
True
```

Segundo segredo

- Há vectores em Haskell.
- Estão no módulo `Array`, que é preciso importar, com `import Array`.
- Os vectores são estruturas de dados sequenciais, como as listas, mas que fornecem acesso em tempo constante a qualquer dos elementos, usando a sua posição.
- As listas só fornecem acesso em tempo constante ao primeiro elemento.

Um cheirinho de vectores

```
a1 = array (1,7) [(1,21),(2,5),(3,6),(5,44),(7,2),(6,12)]
a2 = array (0,5) (zip [0..] ["lisboa","faro","coimbra","aveiro","porto","setubal"])
s1 = [5,3,6,7,2,8,4]
```

Isto aqui é uma
lista, claro, não
um vector.

```
Main> a1 ! 2
5
Main> a1 ! 3
6
Main> a1 ! 4
Program error: undefined array element

Main> a1 ! 5
44
Main> a2!4
"porto"
Main> a2!0
"lisboa"
Main> a2!6
"
Program error: Ix.index: index out of range

Main> s1!!3
7
Main> s1!!10
Program error: Prelude.!!: index too large
```

Terceiro segredo

- Há mónadas.
- Há quê???
- Mónada é um conceito matemático da teoria das categorias.
- Tecnicamente, em Haskell, Monad é uma classe de tipos parametrizados.
- O tipo Maybe (lembra-se?) também é um tipo parametrizado, mas não é da classe Monad.
- O tipo IO é uma instância da classe Monad.
- A notação **do** usa-se só com tipos monádicos.

return ()