Exercícios de Princípios de Programação

Vasco T. Vasconcelos & João Pedro Neto
Universidade de Lisboa – Faculdade de Ciências
Departamento de Informática

2016/2017

I Começando pelo princípio

Tópicos: Expressões, algumas funções pré-definidas (numéricas, etc), as minhas primeiras funções, introdução às listas, algumas funções pré-definidas sobre listas, intervalos, listas infinitas, listas em compreensão, tuplos.

1. Escreva funções que recebam três inteiros e que devolvam:

a) a sua soma.

b) a sua soma se forem todos positivos e zero caso contrário.

2. Escreva uma função que receba três inteiros e devolva **True** se a diferença entre os dois primeiros for inferior ao terceiro e **False** caso contrário.

3. Escreva uma função addDigit que receba dois inteiros, o segundo dos quais entre 0 e 9, e que devolva o inteiro resultante de acrescentar o segundo no fim do primeiro. Por exemplo:

```
ghci> addDigit (-123) 4
-1234
```

4. Quantos elementos tem cada uma das seguintes listas?

a) ['a','b']
b) [['a','b']]
c) [['a','b'], ['c','d']]
d) [[['a','b']]]
e) []
f) [[]]

g) [[],[]]

5. Escreva as seguintes funções.

a) Uma função que devolve **True** se uma dada lista tem mais que 10 elementos, **False** caso contrário.

b) Uma função que verifica se uma lista não está vazia.

c) Uma função que retira o primeiro e último elemento de uma string .

d) Uma função que devolve o segundo elemento de uma lista.

e) Uma função que devolve o penúltimo elemento de uma lista.

f) Uma função que inverte todos os elementos de uma lista excepto o primeiro. O primeiro elemento da lista permanece na primeira posição.

g) Uma função que calcula o somatório dos primeiros 5 elementos de uma lista.

h) Uma função que calcula o somatório dos primeiro n elementos de uma lista. Reescreva a função da alínea anterior utilizando este resultado.

6. Escreva uma função que divide o intervalo entre dois valores em n partições iguais. O resultado deverá ser uma lista de n+1 elementos onde a primeira partição é dada pelos primeiros 2 elementos da lista, a segunda partição pelo segundo e terceiro elementos da lista, e por aí fora. Por exemplo:

```
ghci> partition 10 20 4
[10.0,12.5,15.0,17.5,20.0]
```

7. Determine o valor de cada expressão.

a) [2∗x | x <− [1,2,3]]
b) [x^2 | x <− [1..8], x 'mod' 2 == 0]
c) [x | x <− ['6'..'S'], isDigit x]
d) [(x,y) | x <− [1..3], odd x, y <− [1..3]]
e) [(x,y) | x <− [1..3], y <− [1..3], odd x]

8. Utilizando uma lista em compreensão escreva uma expressão que calcule a soma $1^2 + 2^2 + ... + 100^2$ dos quadrados dos primeiros 100 inteiros.

9. Dizemos que triplo (x, y, z) é Pitagórico se $x^2 + y^2 = z^2$. Utilizando uma lista em compreensão defina a função pithagoreans :: Int −> [(Int, Int, Int)] que calcule a lista de todos os triplos até um dado limite. Por exemplo:

```
ghci> pithagoreans 10
[(3,4,5),(4,3,5),(6,8,10),(8,6,10)]
```

10. Dizemos que um inteiro positivo é *perfeito* se é igual à soma de todos os seus factores, excluindo o próprio número. Utilizando uma lista em compreensão e a função factors :: **Int** −> [**Int**], que devolve os factores do inteiro dado, defina uma função perfects :: **Int** −> [**Int**] que calcula a lista de todos os números perfeitos até um dado limite. Por exemplo:

```
ghci> perfects 500
[6,28,496]
```

11. Defina a lista infinita com todas as potências de dois, powers :: [**Int**].
12. Utilizando uma lista em compreensão defina uma função com assinatura reproduce :: [**Int**] −> [**Int**] que troca cada número n numa lista de inteiros positivos por n cópias dele próprio. Por exemplo:

```
ghci> reproduce [3,5,1]
[3,3,3,5,5,5,5,5,1]
```

Dica: Utilize a função **replicate** onde **replicate** n x é uma lista de comprimento n em que cada elemento é x.
13. Defina uma função pairs de modo a que pairs n seja a lista de todos os pares distintos de números inteiros entre 1 e n.
14. Defina uma função scprod que calcula o produto escalar de dois vectores (aqui um vector é representado por uma lista), ie,

$$scprod(x, y) = \sum_{i=0}^{n} x_i * y_i$$

15. Mostre como uma lista em compreensão com geradores duplos (por exemplo, [(x,y) | x <− [1,2,3], y <− [4,5,6]]) pode ser descrita utilizando apenas geradores simples. Sugestão: utilize a função **concat** e uma compreensão dentro da outra.
16. Quais das seguintes frases são expressões Haskell?

a) ['a','b','c']
b) ('a','b','c')
c) ['a', **True**]
d) [**True**, **False**]
e) ["a disciplina de PP", "eh fixe"]
f) [('a', **False**),('b', **True**)]

g) [**isDigit** 'a', **isLower** 'f', **isUpper** 'h']

h) (['a','b'],[**False**,**True**])

i) [**isDigit**, **isLower**, **isUpper**]

II Tipos e classes de tipos

1. Que tipos usaria para representar:

a) Um ponto tridimensional

b) Um número de 1 a 10

c) Um polígono

d) Um aluno: nome, número e nomes/notas das disciplinas feitas

e) Os alunos de uma turma

f) As palavras de um parágrafo

g) Os parágrafos de um texto

2. Qual o valor das seguintes expressões?

a) "Taprobana"< "As armas e os baroes"

b) "Taprobana"`compare` "As armas e os baroes"

c) **show True** ++ "or " ++ **show False**

d) **show** "As armas e os baroes"

e) **read** "True"

f) **read** (**fst** ("True", 27)) || **False**

g) [**False** .. **True**]

h) [**True** .. **False**]

i) 2 + 3.5

j) (2 :: **Int**) + 3.5

k) **fromIntegral** (2 :: **Int**) + 3.5

3. Seja uma função com tipo (**Ord** a, **Num** b) **=>**(a−>a) −> a −> b. O que ficamos a saber sobre os seus parâmetros e resultado?

4. Qual a assinatura (o tipo mais geral) das seguintes funções?

a) f1 x y = x < y

b) f2 x y z = x == y || z

c) f3 x y z = x == (y || z)

d) f4 x y = **show** x ++ y

e) f5 x y = **show** (x ++ y)

f) f6 x y z = x + y > z

Nota: para obter o tipo de uma qualquer expressão no `ghci` utilize o comando :t <expressão>. Por exemplo :t f1 ou :t **show** ou :t **show** 1.

5. Verdadeiro ou falso?

a) f1 tem tipo **Int** −> **Int** −> **Bool**

b) f1 tem tipo **Integer** −> **Integer** −> **Bool**

c) f1 tem tipo **Int** −> **Integer** −> **Bool**

d) f2 tem tipo [**Char**] −> [**Char**] −> **Bool** −> **Bool**

e) f2 tem tipo [a] −> [a] −> **Bool** −> **Bool**

f) f4 tem tipo **Bool** −> [**Char**] −> [**Char**]

g) f4 tem tipo (**Int** −> **Int**) −> [**Char**] −> [**Char**]

III A sintaxe das funções

1. Usando *pattern matching* escreva funções que devolvam:

a) O primeiro elemento de um par

b) Um dado par com a ordem dos elementos trocados

c) O primeiro elemento de um triplo

d) Um dado triplo com os dois primeiros elementos trocados

e) O segundo elemento de uma lista

f) O segundo elemento do primeiro par de uma lista de pares

Poderia fazê-lo sem *pattern matching*?

2. Qual a diferença entre as seguintes funções?

a) add1 (x,y) = x + y

b) add2 x y = x + y

Escreva a função successor:: **Int** −> **Int** recorrendo a cada uma delas.

3. As funções abaixo diferem? Se sim, como?

a) hd1 (x:_) = x

b) hd2 :: [**Int**] −> **Int**
   hd2 (x:_) = x

c) hd3 :: [a] −> a
   hd3 (x:_) = x

4. Qual a diferença entre as seguintes funções?

a) f1 0 = 0
   f1 x = x−1

b) f2 x = **if** x == 0 **then** 0 **else** x−1

c) f3 x = x−1
   f3 0 = 0

d) f4 x | x /= 0 = x−1
        | **otherwise** = 0

5. Considere a função safetail que se comporta como **tail**, excepto que transforma a lista vazia na lista vazia. Defina safetail utilizando:

a) uma expressão condicional,

b) equações guardadas,

c) *pattern matching*.

6. Mostre como pode definir a disjunção lógica de três modos diferentes, utilizando *pattern matching*. Defina a disjunção como um operador infixo \/. Compare as definições avaliando **True** \/ **undefined** e **False** \/ **undefined**. Nota: **undefined** é uma constante pré-definida que representa uma computação divergente (que não termina). A constante polimórfica **undefined** redunda num erro quando a tentamos avaliar.

7. Utilizando as funções sobre listas constantes no prelude, escreva as seguintes funções.

a) fromTo que obtém a secção de uma lista entre dois índices. Por exemplo:

```
ghci> fromTo 2 4 [0..9]
[2,3,4]
```

   Defina as funções **tail**, **init** e !! usando fromTo.

b) halve :: [a] -> ([a],[a]) que separe em duas sublistas, uma lista de comprimento par. Por exemplo:

```
ghci> halve [1,2,3,4,5,6]
([1,2,3],[4,5,6])
```

8. Qual o tipo mais geral das seguintes funções?

a) second xs = **head** (**tail** xs)

b) swap (x,y) = (y,x)

c) pair x y = (x,y)

d) double x = 2∗x

e) palin xs = **reverse** xs == xs

f) twice f x = f (f x)

9. Encontre o valor e um tipo para cada expressão abaixo.

a) **splitAt** (**length** ['a','b','c','d']) [1..5]

b) **reverse** xs == xs **where** xs = "somos"

c) **tail** (**init** (**tail** (**init** ['a'..'z'])))

d) **take** 2 (**drop** 4 (**replicate** 6 'p'))

e) **zip** xs ys
where xs = **tail** [0,1,2,3]
ys = **init** ['a','b','c','d']

IV Recursão

1. Defina as seguintes funções:

a) **sum**':: **Num** a **=>**[a] −> a, que devolve a soma dos elementos de uma lista.

b) **replicate** ':: **Int** −> a −> [a], que produz uma lista com n elementos idênticos. Se n for negativo, produz a lista vazia.

c) maximo :: **Ord** a **=>**[a] −> a, que devolve o maior elemento de uma lista não vazia.

d) **elem**' :: **Eq** a **=>**a −> [a] −> **Bool**, que decide se um dado elemento existe numa dada lista.

e) substitui :: **Eq** a **=>**a −> a −> [a] −> [a], que substitui o primeiro elemento pelo segundo elemento na lista argumento. Por exemplo:

```
ghci> substitui 0 1 [1,0,3,0,4,0,0]
[1,1,3,1,4,1,1]
```

f) altera :: **Ord** a **=>**[a]−> a −> a −> [a], que substitui todos os elementos da lista argumento que sejam menores que o segundo argumento, pelo terceiro argumento. Por exemplo,

```
ghci> altera [10,0,23,4,14,2,11] 10 5
[10,5,23,5,14,5,11]
```

g) multiplos :: [**Int**] −> **Int** −> [**Int**] que devolve uma lista contendo os elementos de uma dada lista que são múltiplos de um também dado número inteiro. Exemplo:

```
ghci> multiplos [1,3,6,2,5,15,3,5,7,18] 3
[3,6,15,3,18]
```

h) **zip** ':: [a] −> [b] −> [(a,b)], que produz uma lista de pares a partir de duas listas. A lista resultante tem tantos pares quantos o número de elementos da lista argumento mais curta.

i) potencias :: **Int** −> [**Int**] −> [**Int**], que devolve uma lista com potências cuja base é o número dado no primeiro argumento e cujos expoentes são dados pelos valores do segundo argumento. Exemplo:

```
ghci> potencias 3 [1..10]
[3,9,27,81,243,729,2187,6561,19683,59049]
```

j) posicoes :: [**Int**] −> **Int** −> [**Int**] que devolve uma lista contendo as posições dos elementos da lista dada como primeiro argumento que são múltiplos do segundo argumento. Exemplo:

```
ghci> posicoes [1,3,6,2,5,15,3,5,7,18] 3
[1,2,5,6,9]
```

k) frase :: **Int** −> [(**Int**,**String**)]−> **String** que devolve a *string* resultante de concatenar (mantendo a ordem) as *strings* dos pares contidos na lista dada como segundo argumento, que são iguais ao valor do primeiro argumento. Exemplo:

```
ghci> frase 3 [(3,"As "),(1,"Sete ")
  (3,"armas "), (5,"Amor "),
  (3,"e os "), (1,"anos "), (3,"baroes ")]
"As armas e os baroes "
```

l) trocaPares :: [a] −> [a], que troca cada elemento de uma lista com o elemento seguinte, repetindo o processo de par em par de elementos. Se a lista contiver um número ímpar de elementos, o último elemento não é modificado. Exemplo:

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

m) fusao :: (**Ord** a, **Num** b) **=>**[(a,b)] −> [(a,b)] −> [(a,b)], que dadas duas *listas associação*, devolve uma outra lista associação, obtida por junção das duas listas. Uma lista associação é uma lista de pares chave-valor que não contém dois pares com a mesma chave. Neste exercício estamos apenas interessados em listas ordenadas, por ordem crescente dos valores das chave. No caso da função fusao, a lista resultante deve conter tantos pares quantas as chaves distintas existentes nas duas listas argumento e o valor

associado a cada chave será a soma dos valores correspondentes nas duas listas (se cada uma das listas contiver um par com essa mesma chave) ou o valor associado à ocorrência dessa chave no caso de ocorrer somente numa das listas. Exemplo:

```
ghci> fusao [('b',8),('g',2),('m',6),('v',4)]
     [('a',3),('g',5),('m',2)]
[('a',3),('b',8),('g',7),('m',8),('v',4)]
```

2. Um número decimal positivo d pode ser convertido para representação binária através do seguinte algoritmo:

i) se $d < 2$, a sua representação binária é o próprio d;

ii) caso contrário, divide-se d por 2. O resto (0 ou 1) dá-nos o último dígito (o mais à direita) da representação binária;

iii) os dígitos precedentes da representação binária são dados pela representação binária do quociente de d por 2.

Escreva uma função que dado num inteiro devolve a sua representação binária. Por exemplo,

```
ghci> repBinaria 19
10011
```

3. Escreva uma função odioso :: **Int** −> **Bool** que decide se um dado número é um número odioso. Um número odioso é um número não negativo que tem um número ímpar de uns na sua expansão binária. Os primeiros números odiosos são 1, 2, 4, 7, 8, 11.

4. Escreva uma função que recebe dois inteiros i e j e que devolve a representação de i na base j, com $2 \leq j \leq 36$. Esta é uma generalização da função do exercício da representação binária. Para $j > 9$ utilize letras maiúsculas para representar os respectivos dígitos dessas bases ($A = 10$, $B = 11, \ldots$).

5. Programe o seguinte algoritmo de ordenação por inserção em dois passos.

a) Defina uma função **insert** :: **Int** −> [**Int**] −> [**Int**] que insere um inteiro na posição correcta dentro de uma lista *ordenada*. Por exemplo:

```
ghci> insert 3 [1,2,4,5]
[1,2,3,4,5]
```

b) Defina uma função insertSort :: [**Int**] −> [**Int**] que implementa o algoritmo de ordenação por inserção, definido pelas duas regras: (i) a lista vazia está ordenada; (ii) uma lista não vazia pode ser ordenada ordenando a cauda e inserindo a cabeça no resultado.

6. Programe o seguinte algoritmo de ordenação por fusão em dois passos.

a) Defina uma função recursiva merge :: [**Int**] −> [**Int**] −> [**Int**] que funde duas listas *ordenadas*, produzindo uma lista *ordenada*. Por exemplo:

```
ghci> merge [1,3,5] [2,4]
[1,2,3,4,5]
```

b) Defina uma função recursiva mergeSort :: [**Int**] −> [**Int**] que implementa o algoritmo de ordenação por fusão, definido pelas duas regras: (i) listas de comprimento ≤ 1 estão ordenadas; (ii) as outras listas podem ser ordenadas ordenando as suas duas metades e fundindo os resultados.

7. Teste os três algoritmos (incluindo o quicksort do livro de texto). Para isso temos de gerar uma lista grande meio desordenada. A função randomList prepara uma lista de comprimento arbitrário.

```
randomInfiniteList :: [Int]
randomInfiniteList = iterate f 1234
                     where
                     f x = (1343 * x + 997) `mod` 1001
randomList :: Int -> [Int]
randomList n = take n randomInfiniteList
```

Agora podemos fazer os nossos testes.

```
ghci> :set +s
ghci> isort (randomList 1000)
ghci> msort (randomList 1000)
ghci> qsort (randomList 1000)
```

De notar que este exercício apenas dá uma ideia da complexidade em termos de espaço e de tempo dos três algoritmos.

V Funções de ordem superior

1. Descreva o comportamento e um tipo para cada uma das seguintes secções.

a) (∗2)
b) (>0)
c) (1/)
d) (/2)
e) (+1)
f) (++"\n")

2. Determine um tipo e o valor para cada expressão.

a) **map** (+1) [1..3]
b) **map** (>0) [3,−5,−2,0]

c) **filter** (>5) [1..6]

d) **filter** **even** [1..10]

e) **filter** (>0) (**map** (^2) [−3..3])

f) **map** (^2) (**filter** (>0) [−3..3])

g) **map** (++"s") ["A", "arte", "do", "aluno"]

h) **map** ("s"++) ["O", "aluno","bem−comportado"]

i) **map** (**map** (\x −> x∗x)) [[1,2],[3,4,5]]

3. Defina a função **zipWith**' :: (a−>b−>c) −> [a] −> [b] −> [c] semelhante à função **zip** mas que aplica uma dada função a cada par de valores.
4. A função **takeWhile** é semelhante à função **take** com a diferença que espera, como primeiro argumento, um predicado em vez de um inteiro. O valor de **takeWhile** p xs é o mais longo segmento inicial de xs cujos elementos verificam p. Por exemplo:

```
ghci> takeWhile even [4,2,6,1,8,6,2]
[4,2,6]
```

Defina por recursão a função **takeWhile**.
5. Defina a função dropUntil :: (a −> **Bool**) −> [a] −> [a] que elimina os primeiros elementos da lista até que um deles satisfaça a condição dada. Exemplo:

```
ghci> dropUntil (>0) [-4, 0, -8, 3, -2, -5, 3]
[3, -2, -5, 3]
```

6. Defina a função **dropWhile** :: (a −> **Bool**) −> [a] −> [a], que elimina os primeiros elementos da lista enquanto o predicado se verificar.
7. Defina a função aplica :: [a−>a] −> [a] −> [a] que dada uma lista de funções e uma lista de elementos, devolve a lista resultante de aplicar sucessivamente as funções da lista de funções aos valores da lista argumento. Exemplo, onde `5` resulta de multiplicar `1` por `2` e em seguida somar-lhe `3`:

```
ghci> aplica [(*2), (+3)] [1,3,0,4]
[5,9,3,11]
```

8. Defina uma função total :: (**Int** −> **Int**) −> **Int** −> **Int**, de modo a que total f é a função que, no ponto n, retorna f 0 + f 1 + ... + f n.
9. Determine o valor e um tipo das seguintes expressões lambda.

a) (\x −> x + 1)

b) (\x −> x + 1) 6

c) \x −> x > 0

d) \x y −> x + y

e) (\x y −> x + y) 7

f) (\x y −> x + y) 7 3

g) \x −> (\y −> x + y)

h) \f x −> f (f x)

i) (\f x −> f (f x)) (\y −> y + 1)

10. Escreva a função mult x y z = x ∗ y ∗ z utilizando uma expressão lambda.
11. Escreva as secções (++), (++[1,2]), ([1,2]++) como expressões lambda. Quais os seus tipos?
12. Utilizando uma expressão lambda, escreva uma função isNonBlank com a assinatura **Char** −> **Bool** que devolve **True** apenas quando aplicada a caracteres não brancos, isto é, para caracteres que não pertencem à lista [' ','\t','\n'].
13. Dada uma função f do tipo a−>b−>c e dois parâmetros de tipos a e b, escreva uma expressão lambda (a −> b −> c) −> b −> a −> c que se comporta como f, mas que aceita os seus argumentos por ordem inversa. Como aplicação, escreva uma função $x//y = (x/y)^{-1}$.
14. Defina as funções **curry** '::((a,b)−>c)−>a−>b−>c e **uncurry**'::(a−>b−>c)−>(a,b)−>c que, respectivamente, currifica uma função não currificada e descurrifica uma função currificada. Por exemplo,

```
mult (x,y) = x*y -- função não currificada

(*!) = curry' mult

soma = uncurry (+)

ghci> 3 *! 4
12
ghci> soma (3,4)
7
```

15. Determine um tipo e o valor para cada expressão.

a) **foldr** (\y z −> y∗3 + z) 0 [1..4]

b) **foldr** (\x y −> **if** x>0 **then** x+y **else** y) 0 [4,−3,2,−1]

c) **foldr** (\x y −> x^2 + y) 0 [2..5]

d) **foldr** (∗) 1 [−3..(−1)]

e) **foldr** (\x s −> **if** x=='z' **then** x:s **else** s) [] "Oz alunoz dze PzPz"

16. Apresente definições para **map** e **filter** recorrendo à função **foldr**.
17. Defina a função aplica de um exercício acima utilizando uma das variantes da função fold.
18. Escreva um conversor binário para decimal utilizando uma das variantes da função fold. O número binário é apresentado por uma lista de inteiros. Por exemplo:

```
ghci> binary2decimal [1,1,0,1]
13
```

19. Um polinómio pode ser representado por uma lista de coeficientes. Por exemplo, a lista [5,2,0,1,2] representa o polinómio $5x^4 + 2x^3 + 0x^2 + 1x^1 + 2x^0 = 5x^4 + 2x^3 + x + 2$. Defina uma função poly :: **Int** −> [**Int**] −> **Int** que, dado um valor para x e um polinómio, calcule o valor do polinómio nesse ponto.
20. Defina a função selectApply :: (a −> b) −> (a −> **Bool**) −> [a] −> [b] que devolve uma lista contendo os resultados de aplicar a função dada no primeiro argumento aos elementos da lista dada no terceiro argumento, que satisfaçam a condição dada no segundo argumento. Exemplo:

```
ghci> selectApply (*3) (>0) [-4..4]
[3,6,9,12]
```

21. Qual o tipo mais geral de **map map**? e o de **map** . **map**? Utilizando este último, escreva uma função gz :: [[**Int**]] −> [[**Bool**]] que transforme uma matriz (lista de listas) de inteiros numa matriz de valores lógicos, onde cada entrada indica se o valor inicial era ou não maior do que zero. Por exemplo:

```
ghci> gz [[1,2,3],[2,-1,3,7]]
[[True,True,True],[True,False,True,True]]
```

22. Utilizando o operador de composição, defina a função iter que, dado uma função f e um número natural n, devolve a função f^n, i.e., a função f aplicada a si mesma n vezes. Faça uma resolução recursiva e uma outra usando o **foldr**.
23. A função **filter** ' pode ser definida em termos de ., **concat**, e **map**, da seguinte forma:

```
filter' :: (a -> Bool) -> [a] -> [a]
filter' p = concat . map box
            where
            box x = ...
```

Dê uma definição para a função box.
24. Defina a função sumlen que recebe uma lista de inteiros e devolva um par cuja primeira componente é a soma da lista, e a segunda componente é o comprimento da lista. Use a função **foldr**.

VI Módulos

1. Escreva um módulo que implemente um conjunto ordenados de elementos de um tipo instância da classe **Ord**. Como representação do conjunto use listas ordenadas sem elementos repetidos. Considere as seguintes operações:[1]

- empty, criação de um conjunto vazio,
- singleton, criação de de um conjunto singular,
- **insert**, inserção de um elemento num conjunto,
- **union**, união de dois conjuntos,
- intersection, intersecção de dois conjuntos,
- **null**, o dado conjunto é vazio?
- size, o número de elementos no conjunto,
- member, está um elemento em um conjunto?
- **filter** , filtrar os elementos que satisfazem o predicado,
- **partition**, dividir o conjunto em dois, aqueles que satisfazem o predicado e aqueles que não.

2. Escreva um módulo que represente um mapa: uma estrutura de dados que mantém associações entre chaves e valores. Como representação utilize *listas de associação*, isto é, listas de pares chave-valor onde as chaves não devem aparecer repetidas. Prepare operações para as seguintes operações:[2]

- empty :: [(k, a)], o mapa vazio,
- singleton :: k -> a -> [(k, a)], construir um mapa com um único elemento,
- **insert** :: **Ord** k **=>** k -> a -> [(k, a)] -> [(k, a)], juntar uma entrada (chave, valor) ao mapa, substituindo o valor caso a chave já esteja no mapa,
- **null** :: [(k, a)] -> **Bool**, está o mapa vazio?
- size :: [(k, a)] -> **Int**, o número de elementos no mapa,

[1] Muitas outras operações interessantes no módulo Data.Set.

[2] Encontra muitas outras operações interessantes no módulo Data.Map.

- member :: **Eq** k **=>**k −> [(k, a)] −> **Bool**, está a chave no mapa?
- **lookup** :: **Eq** k **=>**k −> [(k, a)] −> **Maybe** a, procurar uma chave no mapa, obtendo o valor associado (**Just** valor), ou **Nothing**, caso contrário,
- **delete** :: **Ord** k **=>**k −> [(k, a)] −> [(k, a)], apagar uma chave e o seu valor de um mapa,
- unionWith :: **Ord** k **=>** (a −> a −> a) −> [(k, a)] −> [(k, a)] −> [(k, a)], união de dois mapas, utilizando uma função para combinar os valores de chaves duplicadas,

VII Construção de tipos e de classes de tipos

1. Defina um tipo de dados que descreva as seguintes formas geométricas: círculo, rectângulo e triângulo. Escreva funções para calcular o perímetro e para verificar se uma figura é regular (uma forma é regular se todos os seus ângulos são iguais e todos os lados são iguais). Torne o tipo de dados instância da classe **Eq**.
2. Utilizando o tipo de dados

```
data Tree a = EmptyTree | Node (Tree a) a (Tree a)
```

escreva as funções abaixo.

a) size :: Tree a −> **Int**, o número de nós na árvore.

b) depth :: Tree a −> **Int**, a profundidade da árvore. A profundidade de uma árvore vazia é zero; aquela de uma árvore não vazia é um mais o máximo das profundidades das sub árvores.

c) flatten :: Tree a −> [a], a lista dos elementos da árvore visitados pelo percurso prefixo. O percurso prefixo de uma árvore não vazia visita primeiro o elemento do nó, depois a sub árvore esquerda e finalmente a sub árvore direita.

d) isFull :: Tree a −> **Bool**, a árvore é cheia? Uma árvore vazia é considerada cheia. Uma árvore não vazia diz-se cheia se as duas sub árvores estão cheias e têm o mesmo número de nós. Numa primeira fase, resolva este exercício recorrendo à função size. Analise a sua complexidade. Desenhe depois uma solução que não percorra a árvore mais do que uma vez.

e) invert :: Tree a −> Tree a, a árvore onde cada sub árvore esquerda é trocada pela sub árvore direita.

f) makeTree :: [a] −> Tree a, a árvore sintetizada a partir de uma lista de elementos da seguinte forma: a cabeça da lista é a raiz da árvore. Dos restantes elementos, a 1ª metade constrói recursivamente a sub árvore da esquerda e a 2ª metade a sub árvore da direita.

3. Reescreva as funções acima recorrendo à função fold.

```
fold :: b −> (b −> a −> b −> b) −> Tree a −> b
fold e _ EmptyTree     = e
fold e f (Node l x r) = f (fold e f l) x (fold e f r)
```

4. Torne o tipo de dados Tree instância da classe **Eq**. Para efeitos deste exercício duas árvores são iguais se contiverem os mesmos elementos.

5. Torne o tipo de dados Tree instância da classe **Show**. A conversão de uma árvore numa **String** deverá ser tal que a árvore

```
Node (Node EmptyTree "cao" (Node EmptyTree "gato"
  EmptyTree)) "peixe" (Node EmptyTree "pulga" EmptyTree)
```

seja convertida em

```
"peixe"
  "cao"
    Empty
    "gato"
      Empty
      Empty
  "pulga"
    Empty
    Empty
```

6. Para o exercício sobre conjuntos da secção anterior, escreva um tipo de dados apropriado. Reescreva depois o módulo de modo a esconder a estrutura do tipo de dados.

7. Para o exercício sobre mapas da secção anterior, escreva um tipo de dados Map apropriado. Reescreva depois o módulo de modo a esconder a estrutura do tipo de dados. Torne o tipo Map instância da classe **Show**. Ao mapa com duas entradas ("a",1) e ("b",2) deve corresponder a *string* {"a": 1, "b": 2}. Junte as funções

a) fromList :: **Ord** k **=>** [(k, a)] −> Map k a, para criar um mapa com os pares dados,

b) toList :: Map k a −> [(k, a)], para obter uma lista com os pares dados constantes num mapa.

8. Considere a classe Visible definida da seguinte forma:

```
class Visible a where
  toString  :: a -> String
  dimension :: a -> Int
```

Crie instâncias desta classe para os tipos **Char**, **Bool**, lista de Visible e pares de Visible.

9. Complete as seguintes declarações.

```
instance (Ord a, Ord b) => Ord (a,b) where ...
instance Ord a => Ord [a] where ...
```

10. Considerando o tipo de dados Nat:

```
data Nat = Zero | Succ Nat
```

escreva as seguintes funções:

a) add :: Nat -> Nat -> Nat, a soma de dois naturais,

b) monus :: Nat -> Nat -> Nat, a subtracção natural, i.e., se o 2º natural for maior que o 1º, a função é igual a zero,

c) **pred** :: Nat -> Nat, calcule o predecessor do natural dado. Esta função deverá estar indefinida para o natural zero,

d) sub :: Nat -> Nat -> Nat, a diferença de dois naturais. Use a função **pred**. Esta função deverá estar indefinida para valores do primeiro argumento menores que o segundo argumento,

e) mult :: Nat -> Nat -> Nat, o produto de dois naturais. Utilize a função add,

f) pot :: Nat -> Nat -> Nat, a potência do 1º natural elevado ao 2º natural,

g) fact :: Nat -> Nat, o factorial do natural dado,

h) remnat :: Nat -> Nat -> Nat, o resto da divisão inteira entre o 1º e o 2º natural,

i) quotnat :: Nat -> Nat -> Nat, o quociente da divisão inteira entre o 1º e o 2º natural,

j) lessThan :: Nat -> Nat -> **Bool**, verifica se o 1º natural é menor que o 2º natural.

VIII Entrada e saída

1. Escreva uma função writePrimes :: [**Int**] -> **IO** () que escreve no `stdout` números primos a partir de uma lista de números naturais. Os números constantes na lista representam as ordens dos números primos. Por exemplo:

```
ghci> writePrimes [7,78,1453,0]
7th prime is 19
78th prime is 401
1453th prime is 12149
0th prime is 2
```

Para gerar a lista de todos os números primos utilize a seguinte função.

```
primes :: [Integer]
primes = sieve [2..]
    where sieve (p:xs) =
        p : sieve [x | x <- xs, x `mod` p > 0]
```

2. Considere a seguinte função.

```
palindrome :: String -> Bool
palindrome xs = xs == reverse xs
```

a) Escreva um programa que lê uma linha do `stdin` e escreve no `stdout` "Sim" ou "Não" conforme a frase é ou não um palíndromo.

b) Escreva um programa que lê continuamente uma linha e que escreve "Sim" ou "Não" conforme a frase é ou não um palíndromo. O programa termina com a introdução de uma linha vazia.

c) Mesmo exercício mas o programa termina com o caracter de fim de ficheiro.[3] Utilize a função **interact** :: (**String** -> **String**) -> **IO** ().

3. Escreva um programa que classifique cada número contido numa lista com "Par" ou "Impar". A classificação deverá aparecer no `stdout`. Exemplo:

```
ghci> showParity [1..4]
Impar
Par
Impar
Par
```

a) Comece por escrever uma função printEven :: **Int** -> **IO**() que escreva no `stdout` "Par" ou "Impar".

b) Escreva a função showParity :: [**Int**] -> **IO**() recursivamente.

c) Mesmo problema utilizando a função **mapM_**, incluída no módulo Data.**List**.

[3] Em Unix: `Ctrl-D`. Pode também utilizar `Ctrl-C` para terminar o processo (Unix e Windows).

4. Escreva um programa que leia um menu de um ficheiro e que imprima no `stdout` as entradas do menu numeradas. Exemplo:

```
$ cat menu.txt
Bacalhau à Gomes Sá
Ensopado de borrego
$ ./menu
1 - Bacalhau à Gomes Sá
2 - Ensopado de borrego
```

a) Comece por escrever uma função linhasComNumeros :: [**String**] −> [**String**] que coloque um número à esquerda de cada linha. Utilize a função **zipWith** :: (a −> b −> c) −> [a] −> [b] −> [c].

b) Escreva o programa utilizando a função **sequence**.

c) Resolva agora o problema utilizando a função **mapM**, incluída no módulo Data.**List**.

5. Implemente o seguinte jogo: o utilizador pensa num número entre 1 e um dado valor. Usando busca binária, e até acertar, o programa sugere um número e o utilizador responde se esse número é <, > ou = ao número em que pensou. No fim o programa indica o número de tentativas que foram necessárias para adivinhar. Implemente o jogo utilizando uma função guess :: **Int** −> **IO** (). Exemplo:

```
ghci> guess 300
150? <
75? <
37? >
56? <
46? =
Sucesso apos 5 tentativas
```

6. Considere o jogo da forca: o programa pede ao primeiro jogador uma palavra. O segundo jogador tenta adivinhar a palavra, letra a cada vez. Exemplo:

```
$ ghc --make forca.hs
[1 of 1] Compiling Main         ( forca.hs, forca.o )
Linking forca ...
$ ./forca
Primeiro Jogador, pense numa palavra:
-----
Segundo Jogador, tente adivinhar:
```

```
Tentativa 1: a
-a-a-
Tentativa 2: p
-a-a-
Tentativa 3: t
-ata-
Tentativa 4: s
-ata-
Tentativa 5: n
nata-
Tentativa 6: l
natal
Acertou!
```

Para evitar mostrar caracteres no ecrã enquanto lê o segredo pode utilizar a seguinte função (sucesso não garantido em Windows).

```
import System.IO
pedeSegredo :: IO String
pedeSegredo = do
    hSetEcho stdin False
    palavra <- getLine
    hSetEcho stdin True
    putStrLn (replicate (length palavra) '-')
    return palavra
```

a) Comece por escrever um programa no qual o segundo jogador dispõe de um número ilimitado de tentativas.

b) Mesmo problema, mas agora o jogador dispõe de um máximo de seis tentativas (cabeça, tronco e quatro membros).

IX Tratamento de ficheiros

1. Escreva uma função

```
toFile :: Show a => FilePath -> [a] -> IO ()
```

que escreva uma lista de elementos (de tipo pertencente à classe **Show**) num ficheiro, colocando um elemento por linha.

2. Escreva uma função

```
fromFile :: Read a => FilePath -> IO [a]
```

que leia um lista de elementos (de tipo pertencente à classe **Read**) de um ficheiro. Cada elemento ocupa uma linha distinta no ficheiro.

3. Escreva uma função

```
sumInts :: FilePath -> IO Int
```

que calcule a soma dos inteiros contidos num dado ficheiro. Cada inteiro ocupa uma linha distinta no ficheiro.

4. Escreva uma função

```
mergeFiles :: (Read a, Ord a) => FilePath -> FilePath -> IO [a]
```

que, dados dois ficheiros *ordenados*, produza a lista ordenada de todos os elementos constantes nos ficheiros. Cada elemento ocupa uma linha distinta no ficheiro. Considere dada uma função `merge :: Ord a => [a] -> [a] -> [a]` que, dadas duas listas ordenadas, produz uma nova lista, também ordenada, com os elementos das duas listas.

5. Considere a seguinte assinatura.

```
filterFiles :: (String->Bool) -> FilePath -> FilePath -> IO ()
```

a) Escreva uma função que leia o conteúdo de um ficheiro, filtre as suas linhas de encontro a um predicado dado, e finalmente escreva o resultado num segundo ficheiro, linha a linha.

b) Escreva uma função

```
filterPrefix :: String -> FilePath -> FilePath -> IO ()
```

que escreva num ficheiro as entradas que comecem com um dado prefixo. Considere dada uma função `isPrefix :: Eq a => [a] -> [a] -> Bool`.

c) Mesmo exercício mas agora os três parâmetros são lidos da linha de comandos.

6. Considere que cada carta de um baralho é representada por um número inteiro entre 1 e 52.

a) Utilizando a função **mkStdGen**, escreva uma funcão que devolva uma mão de cartas de jogar, composta por um dado número de cartas.

```
mao :: Int -> [Int]
```

b) Mesmo exercício mas agora utilizando a função **getStdGen**.

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

X Avaliação Preguiçosa e Listas Infinitas

1. Qual a diferença entre avaliação preguiçosa (*lazy evaluation*) e avaliação ambiciosa (*eager evaluation*) no Haskell? Explique o mecanismo de avaliação preguiçosa do Haskell utilizando, para exemplificação, a expressão:

a) **takeWhile** (>0) (**map** ($-2*$) [-4,-2,0,2])

b) **takeWhile** (<0) (**map** ($*2$) [-4..])

2. A função `until :: (a -> Bool) -> (a -> a) -> a -> a` é tal que `until p f x` devolve o primeiro elemento da sequência `x, f x, f f x, f f f x ...` que verifica o predicado `p`. Utilize a função `until` para descobrir a menor potência de 2 superior a 5798. Defina a função until.

3. Defina as seguintes listas infinitas:

a) potencias :: [**Int**] a lista de potências de 2

b) primos :: [**Int**] a lista dos números primos

c) perfeitos :: [**Int**] a lista dos números perfeitos

d) factoriais :: [**Int**] a lista dos factoriais

4. Defina a função `multiplos :: Int -> [Int]` que, dado um inteiro n, calcula a lista dos múltiplos de n.

5. Mostre, passo a passo, como são avaliadas as expressões:

a) **sum** (**take** 5 [2,5,3,6,9,1,4,7,6,2])

b) **head** (**map** (^2) [-3, 4, -2, 5, 4, 3, 7])

c) **takeWhile** (<20) (**map** (^2) [-3, 4, -2, 5, 3, 7])

d) **take** 4 potencias

e) **take** 4 ((\x $->$ [x ..]) 3)

f) **takeWhile** (\(x,y) $->$ x<20 && y/='c') (**zip** [0..] [' a '..])

6. Escreva uma rede de processos `hammings` que calcula a lista dos *números de Hamming*, por ordem crescente. Os números de *Hamming* são aqueles que têm por factores primos apenas 2, 3 e/ou 5. Por exemplo, 14 não é número de *Hamming* porque $14 = 7 \times 2$ (por ter o factor 7 fica excluído da sequência), enquanto 10 é número de *Hamming* porque $10 = 2 \times 5$.

Utilize a seguinte função `merge` que junta ordenadamente duas listas ordenadas infinitas:

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

Compare a eficiência da sua solução em relação a esta definição mais tradicional:

```
primeFactors :: Integer -> [Integer]
primeFactors n =  primeFactors' n 2
  where
   primeFactors' n m
    | m * m > n     = [n]
    | mod n m == 0 = m : primeFactors' (div n m) m
    | otherwise     = primeFactors' n (m + 1)

isHamming :: Integer -> Bool
isHamming = null . filter (/=5) . filter (/=3)
                 . filter (/=2) . primeFactors

hammings' :: [Integer]
hammings' = filter isHamming [1..]
```

7. Defina uma função que calcule as somas

```
[0, a0, a0+a1, a0+a1+a2, ...
```

dos prefixos de uma lista

```
[a0, a1, a2, ...
```

XI Raciocínio sobre programas

1. Escreva uma especificação para as seguintes funções do **Prelude**, baseada directamente nas duas funções que revelam a constituição de uma lista: **length** e (!!) .

a) **last** :: [a] -> a
b) (++) :: [a] -> [a] -> [a]
c) **tail** :: [a] -> [a]
d) **take** :: **Int** -> [a] -> [a]
e) **drop** :: **Int** -> [a] -> [a]
f) **init** :: [a] -> [a]
g) **map** :: (a -> b) -> [a] -> [b]
h) **zip** :: [a] -> [b] -> [(a,b)]

i) **unzip** :: [(a,b)] −> ([a],[b])

Consegue exprimir a especificação de alguma das funções em termos das outras?

2. Escreva uma especificação para a função **filter** :: (a −> **Bool**) −> [a] −> [a].

3. Quais das seguintes equivalências são verdadeiras para todo o xs?

a) [] : xs == xs
b) [] : xs == [[], xs]
c) xs : [] == xs
d) xs : [] == [xs]
e) xs : xs == [xs,xs]
f) [[]] ++ xs == xs
g) [[]] ++ xs == [xs]
h) [[]] ++ xs == [[], xs]
i) [[]] ++ [[]] == [xs]
j) xs ++ [] == [xs]
k) [xs] ++ [xs] == [xs,xs]

4. Mostre, recorrendo à definição, os seguintes resultados:

a) **length** [x] == 1
b) [x] ++ xs == x:xs
c) **reverse** [x] == [x]

5. Mostre que o operador ++ é associativo e que tem [] como elemento neutro.

6. Mostre por indução as seguintes equivalências:

a) **length** (**reverse** xs) == **length** xs
b) **reverse** (xs ++ ys) == **reverse** ys ++ **reverse** xs
c) **concat** (xss ++ yss) == **concat** xss ++ **concat** yss
d) **sum** (xs ++ ys) == **sum** xs + **sum** ys
e) **sum** (**reverse** xs) == **sum** xs

7. Mostre que para todas as listas finitas ps se tem:

```
zip (fst (unzip ps)) (snd (unzip ps)) == ps
```

8. Recorrendo ao princípio da extensionalidade, mostre que o operador composição, ., é associativo e que tem a função **id** (isto é \x −> x) como elemento neutro.

9. Mostre as seguintes leis sobre **take** e **drop**:

a) **take** m . **take** n == **take** (m ‘**min**‘ n)

b) **drop** m . **drop** n == **drop** (m + n)

c) **take** m . **drop** n == **drop** n . **take** (m + n)

10. Mostre as seguintes leis sobre **map** e **filter** :

a) **map** (f.g) == **map** f . **map** g

b) **map** f . **tail** == **tail** . **map** f

c) **map** f . **reverse** == **reverse** . **map** f

d) **map** f . **concat** == **concat** . **map** (**map** f)

e) **filter** p . **map** f == **map** f . **filter** (p.f)

f) **filter** p . **filter** q == **filter** (p ‘**and**‘q)

g) **filter** p . **concat** == **concat** . **map** (**filter** p)

XII Teste de funções com QuickCheck

1. Dada a função **reverse** :: [a] −> [a], escreva testes que verifiquem se

a) o comprimento da lista de entrada e de saída coincidem,

b) a inversa da inversa é a lista original,

c) a lista inversa é uma permutação da lista original,

d) o i-ésimo elemento da lista inversa é igual ao $(n - 1 - i)$-ésimo elemento da lista original, onde n é o comprimento da lista original.

Quais destas propriedades caracterizam a função **reverse**?

2. Teste as várias funções descritas no exercício **1** da secção da Recursão (cf. página 7 e seguintes).

3. Torne o tipo de dados formas geométricas, exercício **1** na página 15 uma instância da classe Arbitrary.

4. Escreva testes que verifiquem o módulo conjuntos ordenados, exercício **1** na página 14.

a) Classifique as várias operações em *construtoras* (empty e **insert**) e

b) em *observadoras* (**null**, member, size, . . .).

c) Algumas funções podem ser consideradas como *derivadas*. Por exemplo, singleton x é uma abreviatura de **insert** x empty. Identifique as operações que podem ser consideradas derivadas.

d) Construa um teste para cada par observadora-construtora

e) Construa um teste para cada operação derivada

f) Construa testes que construtora-construtora. O que acontece se inserirmos dois valores em sucessão?

g) Torne o tipo de dados Set instância da classe de tipos Arbitrary.

5. Torne o seguinte tipo numa instância de Arbitrary.

```
data Tree = Null
  | Node Tree Int Tree
  deriving (Eq, Ord, Show)
```

Se as árvores produzidas são demasiado grandes, utilize o operador disponível no Quickcheck, sized :: (Int -> Gen a) -> Gen a, que dado um inteiro limita a dimensão da amostra aleatória, para completar o seguinte código:

```
instance Arbitrary Tree where
    arbitrary = sized tree

tree 0 = do
         node <- ??
         return $ Node Null node Null

tree n = do
         c <- choose (1, 2) :: Gen Int
         case c of
             1 -> return $ Null
             2 -> do  -- make Non-Empty Tree
                  node     <- ??
                  subtree1 <- ??
                  subtree2 <- ??
                  return $ Node subtree1 node subtree2
```