



FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Programação Funcional - Exercícios

DCC-FCUP 2020

Definição de funções simples (Aulas 1 e 2)

1. Considere as seguintes definições de funções:

$$\text{inc } x = x + 1$$

$$\text{dobro } x = x + x$$

$$\text{quadrado } x = x * x$$

$$\text{media } x \ y = (x + y)/2$$

Efectuando reduções passo-a-passo, calcule os valores das expressões seguintes:

- (a) *inc* (*quadrado* 5)
 - (b) *quadrado* (*inc* 5)
 - (c) *media* (*dobro* 3) (*inc* 5)
2. Num triângulo, verifica-se sempre a seguinte condição: a medida de um qualquer lado é menor que a soma da dos outros dois. Complete a definição de uma função *triangulo* a b c = ... que testa esta condição; o resultado deve ser um valor booleano.
3. Escreva uma definição em Haskell duma função para calcular a área *A* de um triângulo de lados *a*, *b*, *c* usando a fórmula de Heron:

$$A = \sqrt{s(s-a)(s-b)(s-c)},$$

onde *s* é metade do perímetro do triângulo.

4. Usando as funções do prelúdio-padrão dadas na primeira aula (*head*, *tail*, *length*, *take*, *drop*, *++*, *reverse*, *!!*, *sum* e *product*), escreva uma função *metades* que divide uma lista de comprimento par em duas com metade do comprimento. Exemplo: *metades* [1,2,3,4,5,6,7,8] = ([1,2,3,4], [5,6,7,8]). Investigue o acontece se a lista tiver comprimento ímpar.
5. (a) Mostre que a função *last* (que seleciona o último elemento de uma lista) pode ser escrita como composição das funções do prelúdio-padrão apresentadas na primeira aula. Consegue encontrar duas definições diferentes?
- (b) Mostre que a função *init* (que remove o último elemento duma lista) pode ser definida analogamente de duas formas diferentes.
6. (a) Escreva uma função *binom* com dois argumentos que calcule o *coeficiente binomial*:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Sugestão: pode exprimir *n!* como *product* [1..*n*].

- (b) Para todas as listas de números *xs* e *ys*, temos que *product* (*xs++ys*) = *product* *xs* * *product* *ys*. Use esta propriedade para re-escrever a definição de forma mais eficiente, eliminando factores comuns entre o numerador e denominador.
7. Considere as definições das funções *max* e *min* do prelúdio-padrão que calculam respectivamente o máximo e o mínimo de dois números:

$$\text{max } x \ y = \text{if } x \geq y \text{ then } x \text{ else } y$$

$$\text{min } x \ y = \text{if } x \leq y \text{ then } x \text{ else } y$$

- (a) Usando condições embricadas, escreva definições de duas funções *max3* e *min3* para calcular, respectivamente, o máximo e o mínimo de três números.

- (b) Re-escreva as funções pedidas nas alíneas anteriores de forma a usar *apenas* a composição de *max* e *min* (i.e. sem condicionais ou guardas).
8. Implemente em Haskell as seguintes funções:
- (a) *maxOccurs* :: *Integer* → *Integer* → (*Integer*, *Integer*) que retorna o máximo de dois inteiros e o número de vezes que ocorre.
 - (b) *orderTriple* :: (*Integer*, *Integer*, *Integer*) → (*Integer*, *Integer*, *Integer*) que ordena o triplo por ordem ascendente.
9. Escreva duas definições da função *classifica* :: *Int* → *String*, respectivamente usando expressões condicionais e guardas, que faz corresponder uma classificação qualitativa a uma nota de 0 a 20:

≤ 9	reprovado
10–12	suficiente
13–15	bom
16–18	muito bom
19–20	muito bom com distinção

10. Escreva uma definição da função lógica ou-exclusivo *xor* :: *Bool* → *Bool* → *Bool* usando múltiplas equações com padrões.
11. Pretende-se implementar uma função *safetail* :: [*a*] → [*a*] que estende a função *tail* do prelúdio de forma a dar a lista vazia quando o argumento é a lista vazia (em vez de um erro). Escreva três definições diferentes usando condicionais, equações com guardas e padrões.
12. Escreva duas definições da função *curta* :: [*a*] → *Bool* para testar se uma lista tem zero, um ou dois elementos, usando:
- (a) a função *length* do prelúdio-padrão;
 - (b) múltiplas equações e padrões.
13. Defina uma função *textual* :: *Int* → *String* para converter um número positivo inferior a um milhão para a designação textual em português. Alguns exemplos:

textual 21 = “vinte e um”
textual 1234 = “mil duzentos e trinta e quatro”
textual 123456 = “cento e vinte e três mil quatrocentos e cinquenta e seis”

Sugestão: Comece por definir funções auxiliares para converter para texto número inferiores a 100 e 1000.

Tipos e classes (Aula 2)

14. Indique tipos admissíveis para os seguintes valores.
- (a) [*'a'*, *'b'*, *'c'*]
 - (b) (*'a'*, *'b'*, *'c'*)
 - (c) [(*False*, *'0'*), (*True*, *'1'*)]

- (d) $([False, True], ['0', '1'])$
- (e) $[tail, init, reverse]$
- (f) $[id, not]$

15. Indique tipos possíveis para f e para g por forma que:

- 1. $f\ (2, 5)$ tem tipo Int ;
- 2. $f\ (g\ 5)$ tem tipo Int ;
- 3. $(f\ g)\ 5$ tem tipo Int ;
- 4. $f\ g\ [1, 2, 3]$ tem tipo $[Int]$;
- 5. $f\ (2, 5)$ tem tipo $[Int \rightarrow Int]$.

16. Diga qual o tipo mais geral de f e g por forma que $head\ (f\ g)\ 5$ tenha o tipo $[Int]$.

17. Indique o tipo mais geral para as seguintes definições; tenha o cuidado de incluir restrições de classes no caso de operações com sobrecarga.

- (a) $segundo\ xs = head\ (tail\ xs)$
- (b) $trocar\ (x, y) = (y, x)$
- (c) $par\ x\ y = (x, y)$
- (d) $dobro\ x = 2 * x$
- (e) $metade\ x = x / 2$
- (f) $minuscula\ x = x \geq 'a' \ \&\&\ x \leq 'z'$
- (g) $intervalo\ x\ a\ b = x \geq a \ \&\&\ x \leq b$
- (h) $palindromo\ xs = reverse\ xs == xs$
- (i) $twice\ f\ x = f\ (f\ x)$

18. Indique exemplos de tipos concretos admissíveis e os tipos mais gerais para cada uma das definições dos exercícios 1 e 2. Tenha o cuidado de incluir apenas as restrições de classe estritamente necessárias.

19. Dê exemplo de funções cuja definição é compatível com os tipos seguintes:

- 1. $Int \rightarrow (Int \rightarrow Int) \rightarrow Int$
- 2. $Char \rightarrow Bool \rightarrow Bool$
- 3. $(Char \rightarrow Char \rightarrow Int) \rightarrow Char \rightarrow Int$
- 4. $Eq\ a \Rightarrow a \rightarrow [a] \rightarrow Bool$
- 5. $Eq\ a \Rightarrow a \rightarrow [a] \rightarrow [a]$
- 6. $Ord\ a \Rightarrow a \rightarrow a \rightarrow a$

20. Diga se a função $f :: (a, [a]) \rightarrow Bool$, pode ser aplicada aos argumentos $(2, [3])$, $(2, [])$ e $(2, [True])$. Nos casos afirmativos quais os tipos dos resultados?

21. Repita o exercício anterior para a função $f :: (a, [a]) \rightarrow a$.

Listas em compreensão (Aula 3)

22. Usando uma lista em compreensão, escreva uma expressão para calcular a soma $1^2 + 2^2 + \dots + 100^2$ dos quadrados dos inteiros de 1 a 100.

23. A constante matemática π pode ser aproximada usando expansão em *séries* (i.e. somas infinitas), como por exemplo:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \dots + \frac{(-1)^n}{2n+1} + \dots$$

(a) Escreva uma função *aprox* :: *Int* → *Double* para aproximar π somando em *n* parcelas da série acima (onde *n* é o argumento da função).

(b) A série anterior converge muito lentamente, pelo são necessário muitos termos para obter uma boa aproximação; escreva uma outra função *aprox'* usando a seguinte expansão para π^2 :

$$\frac{\pi^2}{12} = 1 - \frac{1}{4} + \frac{1}{9} - \dots + \frac{(-1)^k}{(k+1)^2} + \dots$$

Compare os resultados obtidos somado 10, 100 e 1000 termos com a aproximação *pi* pré-definida no prelúdio-padrão.

24. Defina uma função *divprop* :: *Int* → [*Int*] usando uma lista em compreensão para calcular a lista de *divisores próprios* de um inteiro positivo (i.e. inferiores ao número dado). Exemplo: *divprop* 10 = [1, 2, 5].

25. Um inteiro positivo *n* diz-se *perfeito* se for igual à soma dos seus divisores (excluindo o próprio *n*). Defina uma função *perfeitos* :: *Int* → [*Int*] que calcula a lista de todos os números perfeitos até um limite dado como argumento. Exemplo: *perfeitos* 500 = [6, 28, 496]. *Sugestão*: utilize a solução do exercício 24.

26. Defina uma função *primo* :: *Int* → *Bool* que testa primalidade: *n* é primo se tem exactamente dois divisores, a saber, 1 e *n*. *Sugestão*: utilize a função do exercício 24 para obter a lista dos divisores próprios.

27. Usando uma função *binom* que calcula o coeficiente binomial, escreva uma definição da função *pascal* :: *Int* → [[*Int*]] que calcula as primeiras linhas triângulo de Pascal. O *triângulo de Pascal* é constituído pelos valores $\binom{n}{k}$ das combinações de *n* em *k* em que *n* é a linha e *k* é a coluna.

28. Escreva uma função *dotprod* :: [*Float*] → [*Float*] → *Float* para calcular o *produto interno* de dois vectores (representados como listas):

$$\text{dotprod } [x_1, \dots, x_n] [y_1, \dots, y_n] = x_1 * y_1 + \dots + x_n * y_n = \sum_{i=1}^n x_i * y_i$$

Sugestão: utilize a função *zip* :: [*a*] → [*b*] → [(*a*, *b*)] do prelúdio-padrão para “emparelhar” duas listas.

29. Um trio (*x*, *y*, *z*) de inteiros positivos diz-se *pitagórico* se $x^2 + y^2 = z^2$. Defina a função *pitagoricos* :: *Int* → [(*Int*, *Int*, *Int*)] que calcule todos os trios pitagóricos cujas componentes não ultrapassem o argumento. Por exemplo: *pitagoricos* 10 = [(3, 4, 5), (4, 3, 5), (6, 8, 10), (8, 6, 10)].

30. Defina uma função *forte* :: *String* → *Bool* para verificar se uma palavra-passe dada numa cadeia de caracteres é “forte”, ou seja: tem 8 caracteres ou mais e pelo menos uma letra maiúscula, uma letra minúscula e um algarismo.

Sugestão: use a função *or* :: [*Bool*] → *Bool* e listas em compreensão.

Definições recursivas e processamento de listas (Aulas 4 e 5)

31. Defina uma função recursiva em Haskell que dado n calcula 2^n (sem recorrer ao operador de exponenciação).

32. Sem consultar as definições na especificação do prelúdio de Haskell, escreva definições recursivas das seguintes funções:

- (a) $and :: [Bool] \rightarrow Bool$ — testar se todos os valores são *True*;
- (b) $or :: [Bool] \rightarrow Bool$ — testar se algum valor é *True*;
- (c) $concat :: [[a]] \rightarrow [a]$ — concatenar uma lista de listas;
- (d) $replicate :: Int \rightarrow a \rightarrow [a]$ — produzir uma lista com n elementos iguais;
- (e) $(!!) :: [a] \rightarrow Int \rightarrow a$ — selecionar o n -ésimo elemento numa lista;
- (f) $elem :: Eq a \Rightarrow a \rightarrow [a] \rightarrow Bool$ — testar se um valor ocorre numa lista.

Nota: não deve usar os mesmos nomes, para não colidir com as definições no Prelude.

33. Mostre que as funções do prelúdio-padrão *concat*, *replicate* e *(!!)* podem também ser definidas sem recursão usando listas em compreensão.

34. A raiz quadrada inteira de um número positivo n é o maior inteiro cujo quadrado é menor ou igual a n . Por exemplo, para 15 e 16, os resultados são, respectivamente 3 e 4.

- (a) Defina uma função recursiva *leastSquare* que calcula o menor inteiro k tal que $k * k \geq n$.
- (b) Defina uma função não recursiva *isqrt* que calcula a raiz quadrada inteira de n .

35. Defina em Haskell:

- (a) a função *factorial* (recursivamente).
- (b) a função *rangeProduct* que calcula

$$a * (a + 1) * \dots * (b - 1) * b$$

- (c) a função *factorial* usando a função *rangeProduct*.

36. Defina uma função em Haskell para calcular o máximo divisor comum de dois inteiros positivos segundo a seguinte definição:

$$mdc(a, b) = \begin{cases} a & b = 0 \\ mdc(b, a \bmod b) & \text{otherwise} \end{cases}$$

37. A função $nub :: Eq a \Rightarrow [a] \rightarrow [a]$ do módulo *Data.List* elimina ocorrências de elementos repetidos numa lista (“nub” em inglês significa *essência*). Por exemplo: *nub* “banana” = “ban”.

Escreva uma definição recursiva para esta função. Sugestão: use uma lista em compreensão com uma guarda para eliminar elementos numa lista.

38. Escreva uma definição da função $intersperse :: a \rightarrow [a] \rightarrow [a]$ do módulo *Data.List* que intercala um valor entre os elementos numa lista. Exemplo: *intersperse* ‘-’ “banana” = “b-a-n-a-n-a”.

39. Defina uma função $maxFun :: (Integer \rightarrow Integer) \rightarrow Integer \rightarrow Integer$, que tendo como argumentos uma função $f :: Integer \rightarrow Integer$ e um inteiro n , retorne o máximo de $f\ 0, f\ 1, \dots, f\ n$.

40. Defina uma função *anyZero* :: $(Integer \rightarrow Integer) \rightarrow Integer \rightarrow Bool$ que tendo como argumentos uma função $f :: Integer \rightarrow Integer$ e um inteiro n retorna *True* se algum valor de $f\ 0, f\ 1, \dots, f\ n$ é igual a zero e *False* caso contrário.
41. Defina uma função *sumFun* :: $(Integer \rightarrow Integer) \rightarrow Integer \rightarrow Integer$, que tendo como argumentos uma função $f :: Integer \rightarrow Integer$ e um inteiro n retorna $(f\ 0) + (f\ 1) + \dots + (f\ n)$.
42. Ordenação de listas pelo **método de inserção**.
- (a) Escreva definição recursiva da função *insert* :: $Ord\ a \Rightarrow a \rightarrow [a] \rightarrow [a]$ da biblioteca *List* para inserir um elemento numa lista ordenada na posição correcta de forma a manter a ordenação. Exemplo: *insert* 2 [0, 1, 3, 5] = [0, 1, 2, 3, 5].
 - (b) Usando a função *insert*, escreva uma definição também recursiva da função *isort* :: $Ord\ a \Rightarrow [a] \rightarrow [a]$ que implementa *ordenação pelo método de inserção*:
 - a lista vazia já está ordenada;
 - para ordenar uma lista não vazia, recursivamente ordenamos a cauda e inserimos a cabeça na posição correcta.
43. Ordenação de listas pelo **método de seleção**.
- (a) Escreva definição recursiva da função *minimum* :: $Ord\ a \Rightarrow [a] \rightarrow a$ do prelúdio-padrão que calcula o menor valor duma lista não-vazia. Exemplo: *minimum* [5, 1, 2, 1, 3] = 1.
 - (b) Escreva uma definição recursiva da função *delete* :: $Eq\ a \Rightarrow a \rightarrow [a] \rightarrow [a]$ da biblioteca *List* que remove a primeira ocorrência dum valor numa lista. Exemplo: *delete* 1 [5, 1, 2, 1, 3] = [5, 2, 1, 3].
 - (c) Usando as funções anteriores, escreva uma definição recursiva da função *ssort* :: $Ord\ a \Rightarrow [a] \rightarrow [a]$ que implementa *ordenação pelo método de seleção*:
 - a lista vazia já está ordenada;
 - para ordenar uma lista não vazia, colocamos à cabeça o menor elemento m e recursivamente ordenamos a cauda sem o elemento m .
44. Ordenação de listas pelo **método merge sort**.
- (a) Escreva uma definição recursiva da função *merge* :: $Ord\ a \Rightarrow [a] \rightarrow [a] \rightarrow [a]$ para juntar duas listas ordenadas numa só mantendo a ordenação. Exemplo: *merge* [3, 5, 7] [1, 2, 4, 6] = [1, 2, 3, 4, 5, 6, 7].
 - (b) Usando a função *merge*, escreva uma definição recursiva da função *msort* :: $Ord\ a \Rightarrow [a] \rightarrow [a]$ que implementa o método *merge sort*:
 - uma lista vazia ou com um só elemento já está ordenada;
 - para ordenar uma lista com dois ou mais elementos, partimos em duas metades, recursivamente ordenamos as duas partes e juntamos os resultados usando *merge*.

Sugestão: comece por definir uma função *metades* :: $[a] \rightarrow ([a], [a])$ para partir uma lista em duas metades.
45. Escreva uma definição da função *bits* :: $Int \rightarrow [[Bool]]$ que obtém todas as sequências de booleanos do comprimento dado. Exemplo: *bits* 2 = [[False, False], [True, False], [False, True], [True, True]]; note que a ordem das sequências não é importante.
- Sugestão: tente exprimir a função por recorrência sobre o comprimento.

46. Escreva uma função *permutations* :: $[a] \rightarrow [[a]]$ para obter a lista com todas as permutações dos elementos uma lista. Assim, se *xs* tem comprimento *n*, então *permutations xs* tem comprimento $n!$. Exemplo: *permutations* [1,2,3] = [[1,2,3], [2,1,3], [2,3,1], [1,3,2], [3,1,2], [3,2,1]]; note que a ordem das permutações não é importante.

Funções de ordem superior (Aulas 6 e 7)

47. Mostre como a lista em compreensão $[f\ x \mid x \leftarrow xs, p\ x]$ se pode escrever como combinação das funções de ordem superior *map* e *filter*.
48. Sem consultar a especificação do Haskell, escreva definições não-recursivas das seguintes funções do prelúdio-padrão:

- (a) $(++) :: [a] \rightarrow [a] \rightarrow [a]$, usando *foldr*;
- (b) *concat* :: $[[a]] \rightarrow [a]$, usando *foldr*;
- (c) *reverse* :: $[a] \rightarrow [a]$, usando *foldr*;
- (d) *reverse* :: $[a] \rightarrow [a]$, usando *foldl*;
- (e) *elem* :: $Eq\ a \Rightarrow a \rightarrow [a] \rightarrow Bool$, usando *any*.

49. Usando *foldl*, defina uma função *dec2int* :: $[Int] \rightarrow Int$ que converte uma lista de dígitos decimais num inteiro. Exemplo: *dec2int* [2,3,4,5] = 2345.

50. A função *zipWith* :: $(a \rightarrow b \rightarrow c) \rightarrow [a] \rightarrow [b] \rightarrow [c]$ do prelúdio-padrão é uma variante de *zip* cujo primeiro argumento é uma função usada para combinar cada par de elementos. Podemos definir *zipWith* usando uma lista em compreensão:

$$zipWith\ f\ xs\ ys = [f\ x\ y \mid (x,y) \leftarrow zip\ xs\ ys]$$

Escreva uma definição recursiva de *zipWith*.

51. Mostre que pode definir função *isort* :: $Ord\ a \Rightarrow [a] \rightarrow [a]$ para ordenar uma lista pelo método de inserção usando *foldr* e *insert*.
52. Considere a função *rotate*, que produz todas as rotações possíveis de uma lista. Ou seja *rotate* [1, 2, 3] = [[1, 2, 3], [2, 3, 1], [3, 1, 2]].
- (a) Defina a função *shift*, que coloca o primeiro elemento de uma lista no final. Ou seja *shift* [1,2, 3] = [2, 3, 1] e *shift* "eat" = "ate".
 - (b) Utilizando *foldr* e *shift* defina a função *rotate*.
53. As funções *foldl1* e *foldr1* do prelúdio-padrão são variantes de *foldl* e *foldr* que só estão definidas para listas com pelo menos um elemento (i.e. não-vazias). *Foldl1* e *foldr1* têm apenas dois argumentos (uma operação de agregação e uma lista) e o seu resultado é dado pelas equações seguintes.

$$\begin{aligned} foldl1\ (\oplus)\ [x_1, \dots, x_n] &= (\dots (x_1 \oplus x_2) \dots) \oplus x_n \\ foldr1\ (\oplus)\ [x_1, \dots, x_n] &= x_1 \oplus (\dots (x_{n-1} \oplus x_n) \dots) \end{aligned}$$

- (a) Mostre que pode definir as funções *maximum*, *minimum* :: $Ord\ a \Rightarrow [a] \rightarrow a$ do prelúdio-padrão (que calculam, respectivamente, o maior e o menor elemento duma lista não-vazia) usando *foldl1* e *foldr1*.

- (b) Mostre que pode definir *foldl1* e *foldr1* usando *foldl* e *foldr*. Sugestão: utilize as funções *head*, *tail*, *last* e *init*.

54. A função *add* pode ser definida em termos das funções:

```
succ i = i + 1
pred i = i - 1
```

pelas equações:

```
add i 0 = i
add i j = succ (add i (pred j))
```

- (a) Dê uma definição semelhante para *mult* que use apenas *add* e *pred*. Dê uma definição para *exp* que use apenas *mult* e *pred*. Qual será a próxima função nesta sequência?
- (b) A função *foldi* sobre inteiros pode ser definida da seguinte forma:

```
foldi :: (a -> a) -> a -> Integer -> a
foldi f q 0 = q
foldi f q i = f (foldi f q (pred i))
```

Defina as funções *add*, *mult* e *exp* em termos *foldi*.

- (c) Defina as funções *fact* (factorial) e *fib* (Fibonacci) utilizando a função *foldi*.

55. A função de ordem superior *until* :: (a → Bool) → (a → a) → a → a está definida no prelúdio-padrão; *until* p f é a função que repete sucessivamente a aplicação de f ao argumento até que p seja verdade. Usando *until*, escreva uma definição não recursiva da função

$$mdc\ a\ b = \text{if } b == 0 \text{ then } a \text{ else } mdc\ b\ (a \text{ 'mod' } b)$$

que calcula o máximo divisor comum pelo algoritmo de Euclides.

56. A função do prelúdio *scanl* é uma variante do *foldl* que produz a lista com os valores acumulados:

$$scanl\ f\ z\ [x_1, x_2, \dots] = [z, f\ z\ x_1, f\ (f\ z\ x_1)\ x_2, \dots]$$

Por exemplo:

$$scanl\ (+)\ 0\ [1, 2, 3] = [0, 0 + 1, 0 + 1 + 2, 0 + 1 + 2 + 3] = [0, 1, 3, 6]$$

Em particular, para listas finitas xs temos que *last* (scanl f z xs) = foldl f z xs.

Escreva uma definição recursiva de *scanl*; deve usar outro nome para evitar colidir com a definição do prelúdio.

Listas infinitas (Aula 8)

57. Defina as listas infinitas dos factoriais e dos números de Fibonacci,

```
factorial = [1, 1, 2, 6, 24, 120, 720, ...]
fibonacci = [0, 1, 1, 2, 3, 5, 8, 13, 21, ...]
```

58. Defina uma função *merge* de duas listas infinitas ordenadas, que junta as listas retornando uma lista infinita também ordenada. Utilize a função *merge* com as listas de todas as potências de 2 e todas as potências de 3.

Defina a lista de números cujos únicos factores primos são 2,3 e 5 (chamados números de Hamming):

`hamming = [1,2,3,4,5,6,8,9,10,12,15,...`

(sugestão: use a função *merge* definida anteriormente)

59. Defina uma função de somas sucessivas

`somas :: [Int] -> [Int]`

que calcula as somas sucessivas

`[0,n0,n0+n1,n0+n1+n2,...`

a partir de uma lista infinita

`[n0,n1,n2,...`

60. Neste exercício pretende-se definir o triângulo de Pascal completo como uma *lista infinita pascal* :: `[[Int]]` das linhas do triângulo.

- (a) Escreva uma definição de *pascal* usando a função *binom* do Exercício 6. Note que $pascal !! n !! k = binom\ n\ k$, para quaisquer n e k tais que $n > 0$ e $0 \leq k \leq n$.
- (b) Escreva outra definição que evite o cálculo de factoriais usando as seguintes propriedades de coeficientes binomiais:

$$\binom{n}{0} = \binom{n}{n} = 1 \qquad \binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1} \quad (\text{se } n > k)$$

61. Considere as funções *shift* e *rotate* do exercício 52. Redefina a função *rotate*, usando a função *shift* e lista infinita de rotações.
62. Considere a função *stringsN* que, dado um n , calcula a lista de todas as strings de tamanho n compostas por letras minúsculas.
- (a) Comece por definir uma função *strings* que calcula a lista infinita de todas as sequências de letras minúsculas.
 - (b) Utilizando a função da alínea anterior, defina a função *stringsN*.
63. Sem recorrer às funções *map*, *foldl* e *scanl*, escreva uma definição em Haskell para a função *scanl*, que opere sobre listas infinitas. Por exemplo:

`scanl (*) 1 [2,2..] = [1,2,4,8,16,32,64,128,256,512,...]`

Programas interactivos (Aula 9)

64. Escreva uma função *elefantes* :: *Int* → *IO* () tal que, por exemplo, *elefantes* 5 imprime os seguintes versos:

```
Se 2 elefantes incomodam muita gente,  
3 elefantes incomodam muito mais!  
Se 3 elefantes incomodam muita gente,  
4 elefantes incomodam muito mais!  
Se 4 elefantes incomodam muita gente,  
5 elefantes incomodam muito mais!
```

Sugestão: utilize a função *show* :: *Show* a ⇒ a → *String* para converter um inteiro numa cadeia de caracteres; pode ainda re-utilizar a função *sequence_* :: [*IO* a] → *IO* () para executar uma lista de ações.

65. Escreva um programa completo que reproduza a funcionalidade do utilitário *wc* de Unix: ler um ficheiro de texto da entrada-padrão e imprimir o número de linhas, número de palavras e de *bytes*. Exemplo:

```
$ echo "a maria tinha um cordeirinho" | wc  
1      5      29
```

Sugestão: modifique o exemplo apresentado na aula teórica; utilize as funções *words* e *lines* do prelúdio-padrão.

66. Escreva um programa completo que lê linhas de texto da entrada-padrão e imprime cada linha invertida.
67. Escreva um programa completo que codifique a entrada-padrão usando a cifra de César de 13 posições (ver <http://www.rot13.com>). Exemplo:

```
$ echo "a maria tinha um cordeirinho" | ./rot13  
n znevn gvaun hz pbeqrvevaub
```

68. Escreva uma função interactiva *adivinha* :: *String* → *IO* () que implemente um jogo de adivinha duma palavra secreta dada como argumento pelo utilizador; um outro jogador vai tentar adivinhá-la.
- O programa deve mostrar a palavra, substituindo as letras desconhecidas por traços e pedir uma nova letra; todas as ocorrências dessa letra na palavra devem então ser reveladas. O jogo termina quando o jogador adivinha a palavra; o programa deve então imprimir o número de tentativas (ver Figura 1).

69. O jogo *Nim* desenrola-se com cinco filas de peças idênticas (representadas por estrelas), cujo estado inicial é o seguinte:

```
1 : * * * * *  
2 : * * * *  
3 : * * *  
4 : * *  
5 : *
```

Dois jogadores vão alternadamente retirar uma ou mais estrelas de uma das filas; ganha o jogador que remover a última estrela ou grupo de estrelas.

Implemente este jogo como um programa em Haskell que pergunte as jogadas de cada jogador e actualize o tabuleiro. Sugestão: represente estado do jogo como uma lista com o número de estrelas em cada fila; o estado inicial será então [5, 4, 3, 2, 1].

```

-----
? a
-a-a-a
? e
Não ocorre
-a-a-a
? b
ba-a-a
? n
banana
Adivinhou em 4 tentativas

```

Figura 1: Exemplo de interação do jogo de adivinha.

Árvores de pesquisa (Aula 10)

Nos exercícios seguintes considere a definição do tipo de árvores de pesquisa apresentada na aulas teóricas:

$$\text{data } Arv\ a = \text{Vazia} \mid \text{No } a\ (Arv\ a)\ (Arv\ a)$$

70. Escreva uma definição recursiva da função $\text{sumArv} :: \text{Num } a \Rightarrow Arv\ a \rightarrow a$ que soma todos os valores numa árvore binária de números.
71. Baseado-se na função $\text{listar} :: Arv\ a \rightarrow [a]$ apresentada na aula teórica, escreva a definição numa função para listar os elementos numa árvore de pesquisa por *ordem decrescente*.
72. Escreva uma definição da função $\text{nivel} :: \text{Int} \rightarrow Arv\ a \rightarrow [a]$ tal que $\text{nivel } n\ arv$ é a lista ordenada dos valores da árvore no nível n , isto é, a uma altura n (considerando que a raiz tem altura 0).
73. Experimente usar o interpretador de Haskell para calcular a altura de árvores de pesquisa com n valores.
 - (a) usando o método de partições binárias, e.g. $\text{construir } [1..n]$;
 - (b) usando inserções simples, e.g. $\text{foldr inserir Vazia } [1..n]$;
 - (c) usando inserções AVL, e.g. $\text{foldr inserirAVL Vazia } [1..n]$;

Experimente com $n = 10, 100$ e 1000 e compare a altura obtida com o menorante teórico: uma árvore binária com n nós tem altura $\geq \log_2 n$.

74. Escreva uma definição da função de ordem superior $\text{mapArv} :: (a \rightarrow b) \rightarrow Arv\ a \rightarrow Arv\ b$ tal que $\text{mapArv } f\ t$ aplica uma função f a cada valor numa árvore t .
75. Escreva uma definição da função de ordem superior $\text{foldArv} :: (a \rightarrow b \rightarrow b \rightarrow b) \rightarrow b \rightarrow Arv\ a \rightarrow b$ tal que $\text{foldArv } f\ v\ t$ se comporte como a função foldr , mas opere sobre árvores.
76. Neste exercício pretende-se implementar uma variante da remoção de um valor numa árvore de pesquisa simples.
 - (a) Baseando-se na função $\text{mais_esq} :: Arv\ a \rightarrow a$ apresentada na aula teórica, escreva uma definição da função $\text{mais_dir} :: Arv\ a \rightarrow a$ que obtém o valor mais à direita numa árvore (i.e., o maior valor).

- (b) Usando a função da alínea anterior, escreva uma definição alternativa da função *remove* :: *Ord* *a* \Rightarrow *a* \rightarrow *Arv* *a* \rightarrow *Arv* *a* que use o valor mais à direita da sub-árvore esquerda no caso de um nó com dois descendentes não-vazios.
77. Escreva uma definição da função *removeAVL* :: *Ord* *a* \Rightarrow *a* \rightarrow *Arv* *a* \rightarrow *Arv* *a* para remover um valor duma árvore AVL de forma a manter a árvore resultante equilibrada. Sugestão: modifique a função análoga para árvores de pesquisa simples.

Definição de Tipos de Dados (Aula 11)

78. Implemente em Haskell:

- (a) Defina um datatype *Shape* para representar círculos de um determinado raio (*Circle* *Float*) ou rectângulos de lados *a*, *b* (*Rectangle* *Float* *Float*).
- (b) Defina uma função para calcular o perímetro de uma figura geométrica *Shape*.
79. Um rectângulo de lados paralelos aos eixos é definido pelas suas extremidades inferior esquerda e superior direita. Cada uma dessas extremidades é um ponto caracterizado por um par de coordenadas.
1. Caracterize os tipos apropriados para a representação dos rectângulos do tipo indicado.
 2. Defina uma função que calcule a área de um rectângulo.
 3. Defina uma função que indique se dois rectângulos dados se intersectam (i.e. têm pelo menos um ponto em comum).
80. Escreva uma função *parent* :: *String* \rightarrow *Bool* que verifique se uma cadeia de caracteres é uma sequência de parêntesis curvos e rectos correctamente emparelhados; por exemplo:

$$\text{parent } "((((([()])))" = \text{True} \quad \text{parent } "([()]" = \text{False}$$

Sugestão: represente parêntesis abertos usando uma pilha dos caracteres '(', '[' e '{'; utilize o módulo *Stack* apresentado na aula teórica.

81. Na *notação polaca invertida* (abreviado para *RPN* do inglês “*reverse polish notation*”) colocamos cada operador binário após os dois operandos; por exemplo, a expressão $42 \times 3 + 1$ escreve-se “42 3 * 1 +”. Nesta notação não necessitamos de parêntesis ou de precedências entre operadores.

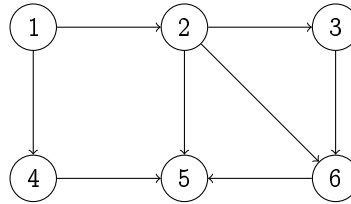
Pretende-se escrever uma função para calcular o valor de uma expressão em RPN; este cálculo pode ser feito percorrendo a expressão uma só vez usando uma pilha para guardar valores intermédios.

- (a) Escreva uma função auxiliar *calc* :: *Stack* *Float* \rightarrow *String* \rightarrow *Stack* *Float* que implemente uma operação (se o 2º argumento for “+”, “*”, “-” ou “/” ou coloque um operando na pilha (se o 2º argumento for um numeral); o resultado deve ser a pilha modificada.
Sugestão: utilize a função *read* :: *String* \rightarrow *Float* do prelúdio-padrão para converter um número em texto para vírgula flutuante.
- (b) Usando a função anterior e o módulo *Stack* apresentados nas aulas teóricas, escreva a função *calcular* :: *String* \rightarrow *Float* que calcula o valor duma expressão em RPN; por exemplo:
calcular “42 3 * 1 +” = 127.
Sugestão: utilize a função *words* :: *String* \rightarrow [*String*] do prelúdio-padrão para partir uma cadeia de caracteres em palavras.

- (c) Escreva um programa principal que leia uma expressão em RPN da entrada padrão como uma cadeia de caracteres da entrada padrão e calcule o seu valor. Experimente o programa com expressões correctas e incorrectas e interprete os resultados.

82. Considere um grafo dirigido $G = (V, E)$ constituído por um conjunto finito de vértices V e por um conjunto finito de arcos E , sendo cada arco (x, y) um par ordenado de vértices.

1. Defina em Haskell uma estrutura apropriada para representar grafos.
2. Represente o grafo da seguinte figura:



3. Escreva uma função cujos dados sejam um grafo dirigido (qualquer!) e dois vértices desse grafo e cujo resultado seja o número de caminhos distintos entre os dois vértices dados. Supõe-se que o grafo não tem ciclos.

Por exemplo, para o grafo da figura e para os vértices 1 e 5 o resultado é 4 (os caminhos correspondentes - que não é necessário calcular!- são $[1, 4, 5]$, $[1, 2, 5]$, $[1, 2, 6, 5]$ e $[1, 2, 3, 6, 5]$).

83. As equações seguintes especificam as operações do tipo abstracto *pilha*:

$$\text{pop } (\text{push } x \ s) = s \quad (1)$$

$$\text{top } (\text{push } x \ s) = x \quad (2)$$

$$\text{isEmpty empty} = \text{True} \quad (3)$$

$$\text{isEmpty } (\text{push } x \ s) = \text{False} \quad (4)$$

A propriedade (1) foi verificada na aula; verifique que a implementação usando listas apresentada na aula teórica satisfaz as restantes propriedades (2)–(4)

84. Escreva um programa para contar as ocorrências de letras num texto. Sugestão: represente a contagem de ocorrências usando um dicionário `Map Char Int`.

85. Modifique o exemplo da contagem de palavras distintas de forma a usar listas sem repetição em vez de `Data.Set` (cuja implementação usa árvores equilibradas).

A análise assintótica diz-nos que, para conjuntos grandes, a pesquisa em árvores equilibradas é mais eficiente do que em listas. Investigue experimentalmente qual o tamanho de texto a partir do qual tal se verifica. Sugestão: use o comando `:set +s` do interpretador para imprimir o tempo usado numa computação.

86. Considere o tipo abstrato *Set* *a* para conjuntos finitos de valores de tipo *a* com as seguintes operações:

$$\text{empty} :: \text{Set } a$$

$$\text{insert} :: \text{Ord } a \Rightarrow a \rightarrow \text{Set } a \rightarrow \text{Set } a$$

$$\text{member} :: \text{Ord } a \Rightarrow a \rightarrow \text{Set } a \rightarrow \text{Bool}$$

Escreva uma implementação deste tipo usando árvores binárias de pesquisa simples.

87. Considere as operações de união, interseção e diferença entre conjuntos; todas estas operações têm o mesmo tipo:

$$\text{union, intersect, difference} :: \text{Ord } a \Rightarrow \text{Set } a \rightarrow \text{Set } a \rightarrow \text{Set } a$$

Acrescente estas operações à implementação que fez para o exercício anterior.

88. Considere o tipo abstrato $\text{Map } k \ a$ para associações entre chaves de tipo k e valores de tipo a com as seguintes operações:

$$\text{empty} :: \text{Map } k \ a$$

$$\text{insert} :: \text{Ord } k \Rightarrow k \rightarrow a \rightarrow \text{Map } k \ a \rightarrow \text{Map } k \ a$$

$$\text{lookup} :: \text{Ord } k \Rightarrow k \rightarrow \text{Map } k \ a \rightarrow \text{Maybe } a$$

Escreva uma implementação deste tipo abstrato usando árvores binárias de pesquisa simples.

Raciocinar sobre programas (Aula 12)

89. Considere a definição recursiva dos naturais e da adição.

$$\text{data Nat} = \text{Zero} \mid \text{Succ Nat}$$

$$\text{Zero} + y = y$$

$$\text{Succ } x + y = \text{Succ } (x + y)$$

Prove a associatividade da adição: $x + (y + z) = (x + y) + z$ para todo x, y, z .

Sugestão: use indução sobre x .

90. Usando indução sobre a lista xs , prove a associatividade da concatenação: $(xs ++ ys) ++ zs = xs ++ (ys ++ zs)$.

Sugestão: a prova é análoga à do exercício anterior.

91. Prove a distributividade de reverse sobre $++$:

$$\text{reverse } (xs ++ ys) = \text{reverse } ys ++ \text{reverse } xs$$

usando indução sobre a lista xs .

Sugestão: será útil usar o resultado do exercício anterior (associatividade de $++$). Tenha em atenção que as listas xs, ys aparecem por ordem contrária no lado direito da igualdade!

92. Considere as definições das funções de ordem-superior map e \circ (composição de duas funções):

$$\text{map } f [] = []$$

$$\text{map } f (x : xs) = f x : \text{map } f xs$$

$$(f \circ g) x = f (g x)$$

Usando indução sobre listas, mostre que $\text{map } f (\text{map } g xs) = \text{map } (f \circ g) xs$.

- 93.** Usando as seguintes definições das funções $take, drop :: Int \rightarrow [a] \rightarrow [a]$ e a definição canónica de $++$, mostre que $take\ n\ xs ++ drop\ n\ xs = xs$.

$$\begin{aligned} take\ 0\ xs &= [] \\ take\ n\ [] \quad | \quad n > 0 &= [] \\ take\ n\ (x : xs) \quad | \quad n > 0 &= x : take\ (n - 1)\ xs \\ drop\ 0\ xs &= xs \\ drop\ n\ [] \quad | \quad n > 0 &= [] \\ drop\ n\ (x : xs) \quad | \quad n > 0 &= drop\ (n - 1)\ xs \end{aligned}$$

Sugestão: use indução sobre n e análise de casos da lista xs .

- 94.** Usando indução sobre listas, prove que $length\ (map\ f\ xs) = length\ xs$, isto é, a função map preserva o comprimento da lista.

- 95.** Usando indução sobre listas, prove que $sum\ (map\ (1+) \ xs) = length\ xs + sum\ xs$. Recorde que a notação $(1+)$ representa a função que adiciona 1 a um número.

- 96.** Usando indução sobre listas, mostre que

$$map\ f\ (xs ++ ys) = map\ f\ xs ++ map\ f\ ys$$

para quaisquer funções f e listas finitas xs e ys .

- 97.** Usando indução sobre listas, mostre que

$$map\ f\ (reverse\ xs) = reverse\ (map\ f\ xs)$$

Sugestão: use a propriedade provada no exercício 96.

- 98.** Considere a seguinte definição da função $inserir :: Int \rightarrow [Int] \rightarrow [Int]$ que insere um valor numa lista crescente de inteiros mantendo a ordenação.

$$\begin{aligned} inserir\ x\ [] &= [x] \\ inserir\ x\ (y : ys) \quad | \quad x \leq y &= x : y : ys \\ inserir\ x\ (y : ys) \quad | \quad x > y &= y : inserir\ x\ ys \end{aligned}$$

Usando indução sobre listas, prove que $length\ (insert\ x\ xs) = 1 + length\ xs$.

- 99.** Considere a declaração dum tipo recursivo para árvores binárias anotadas:

$$data\ Arv\ a = Folha \quad | \quad No\ a\ (Arv\ a)\ (Arv\ a)$$

Usando indução sobre árvores, mostre que o *número de folhas* é sempre mais um do que o *número de nós intermédios*.

Sugestão: comece por definir duas funções recursivas para calcular o número de folhas e nós intermédios numa árvore.

100. Considere a função para listar por ordem os elementos numa árvore binária:

```
listar :: Arv a → [a]
listar Vazia = []
listar (No x esq dir) = listar esq ++ [x] ++ listar dir
```

Empregue a técnica para eliminar concatenações apresentada na aula teórica para derivar uma versão mais eficiente desta função.

Sugestão: sintetize uma definição recursiva da função auxiliar $listarAcc :: Arv a \rightarrow [a] \rightarrow [a]$ tal que $listarAcc\ t\ xs = listar\ t ++ xs$.

101. Considere a seguinte declaração de tipo:

```
data Arv a = Vazia | No a (Arv a) (Arv a)
```

Nota: As seguintes alíneas deverão ser definidas para árvores de valores numéricos.

- (a) Defina uma função soma, que calcule a soma dos valores da árvore dada.
- (b) Defina uma função valorArv, que devolva o valor na raiz da árvore dada, ou zero no caso da árvore ser vazia.
- (c) Defina uma função somasTree, que devolva uma árvore em que cada nó contém as somas parciais dos valores em cada sub-árvore da árvore dada.
Nota: A soma dos elementos em cada sub-árvore deve ser efectuada uma única vez e **sem** recorrer à função soma definida anteriormente.
- (d) Considerando as funções definidas na questão anterior, mostre que para qualquer árvore numérica t , $soma\ t = valorArv\ (somasTree\ t)$.

102. Considere o tipo $Arv\ a$ definido no exercício anterior.

- Defina uma função simetrica, que dada uma árvore, devolva uma nova árvore em que os elementos aparecem em posições simétricas (em relação à raiz) à sua posição inicial.
- Mostre que para qualquer árvore t ,

```
listar t = reverse (listar (simetrica t)).
```

103. Considere a função soma, definida anteriormente.

- (a) Recorde a função foldr definida para listas. Defina uma função foldtree, que se comporte como a função foldr, mas opere sobre árvores. **Nota:** A função foldtree deverá ter como parâmetro uma função do tipo $a \rightarrow b \rightarrow b \rightarrow b$. Por exemplo, podemos definir a função soma t da alínea anterior, como $foldtree\ soma3\ 0\ t$, onde $soma3\ x\ y\ z = x + y + z$.
- (b) Mostre que para qualquer árvore numérica t , $soma\ t = foldtree\ soma3\ 0\ t$, onde $soma3\ x\ y\ z = x + y + z$.

Exercícios suplementares

- 104.** (a) Escreva uma função $mindiv :: Int \rightarrow Int$ cujo resultado é o menor divisor próprio do argumento (i.e. o menor divisor superior a 1). Note que se $n = p \times q$, então p e q são ambos divisores de n ; assim, se $p \geq \sqrt{n}$, então $q \leq \sqrt{n}$ pelo que $mindiv\ n \leq \sqrt{n}$.

(b) Utilize *mindiv* para definir um teste de primalidade mais eficiente do que o exercício 26: n é primo se $n > 1$ e o seu menor divisor próprio for igual a n .

105. A *cifra de César* é um dos métodos mais simples para codificar um texto: cada letra é substituída pela que dista k posições à frente no alfabeto; se ultrapassar a letra Z, volta à letra A. Por exemplo, para $k = 3$, a substituição efectuada é

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C

e o texto “ATAQUE DE MADRUGADA” é transformado em “DWDTXH GH PDGUXJDGD”.

Escreva uma função $cifrar :: Int \rightarrow String \rightarrow String$ para cifrar uma cadeia de caracteres usando um deslocamento dado. Note que $cifrar (-n)$ é a função inversa de $cifrar n$, pelo que não necessitamos de definir uma função especial para descodificação.

106. Um quadrado mágico de dimensão n é uma matriz $n \times n$ contendo os inteiros de 1 até n^2 tal que a soma de qualquer linha, coluna ou diagonal dá um mesmo valor. A figura seguinte representa um quadrado mágico de dimensão 3, em que cada linha, coluna e diagonal soma 15.

6	7	2
1	5	9
8	3	4

Escreva um programa para para testar se uma matriz $[[Int]]$ é um quadrado mágico.

107. O *problema das oito rainhas* consiste em determinar posições para colocar oito rainhas num tabuleiro de Xadrez (com 8×8 casas) de forma a que nenhuma rainha esteja em linha de ataque de outra (i.e. usando movimentos em linhas, colunas ou diagonais).

Escreva um programa para procurar *todas* as soluções do problema das oito rainhas.

108. O *problema de decompor uma quantia em trocos* pode ser formalizado da seguinte maneira: dado um natural n e uma lista de naturais xs , encontrar decomposições de n como soma de valores em xs (eventualmente com repetições).

Por exemplo, para $n = 25$ e $xs = [2, 5, 10]$ uma decomposição possível é $[5, 10, 10]$ (porque $25 = 5 + 10 + 10$). Outras possibilidades são $[5, 5, 5, 5, 5]$ ou $[2, 2, 2, 2, 5, 10]$ (e há mais alternativas).

Escreva uma definição duma função $decompor :: Int \rightarrow [Int] \rightarrow [[Int]]$ tal que $decompor n xs$ encontra *todas* as alternativas para o problema dos trocos. O resultado deverá ser a lista vazia quando o problema não tem solução.

109. O *Sudoku* é um *puzzle* lógico em que o objectivo é completar o preenchimento de uma grelha de 9×9 com inteiros de 1 a 9 de forma que cada uma das 9 linhas, colunas e quadrados 3×3 não contenha números repetidos.

Pretende-se escrever um programa para resolver este tipo de problemas; pode representar a grelha por uma lista de linhas:

type Grid = $[[Int]]$

Cada entrada será um inteiro de 1 a 9 ou 0 se ainda não estiver preenchida.

- (a) Escreva uma definição da função $check :: Grid \rightarrow Bool$ que verifique se uma grelha respeita as condições do problema (i.e. a ausência de repetições nas linhas, colunas e quadrados).

(b) Escreva uma definição da função $solve :: Grid \rightarrow [Grid]$ que obtêm todas soluções de um problema Sudoku; o argumento é uma grelha parcialmente preenchida.

110. Pretende-se que resolva este exercício sem usar *words* e *unwords* do prelúdio-padrão (pois $words = palavras$ e $unwords = despalavras$).

(a) Escreva uma definição da função $palavras :: String \rightarrow [String]$ que decompõe uma linha de texto em palavras delimitadas por um ou mais espaços.

Exemplo: $palavras$ "Abra- ca- drabra!" = ["Abra-", "ca-", "dabra!"].

(b) Escreva uma definição da função $despalavras :: [String] \rightarrow String$ que concatena uma lista de palavras juntando um espaço entre cada uma. Note que $despalavras$ não é a função inversa de $palavras$; encontre um contra-exemplo que justifique esta afirmação.

111. Considere duas séries (i.e. somas infinitas) que convergem para π :

$$\pi = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \dots \quad (5)$$

$$\pi = 3 + \frac{4}{2 \times 3 \times 4} - \frac{4}{4 \times 5 \times 6} + \frac{4}{6 \times 7 \times 8} - \dots \quad (6)$$

Escreva duas funções $calcPi1, calcPi2 :: Int \rightarrow Double$ que calculam um valor aproximado de π usando o número de parcelas dado como argumento; investigue qual das séries converge mais depressa para π .

Sugestão: construa listas infinitas para os numeradores e denominadores dos termos separadamente e combine-as usando *zip/zipWith*.

112. Escreva um programa completo que implemente a funcionalidade do utilitário *cal* de Unix: imprimir o calendário dum ano formatado numa grelha 4×3 meses; a figura seguinte ilustra uma sugestão de formatação de um mês.

```

Fevereiro 2016
Su Mo Tu We Th Fr Sa
    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

```

113. A informação de altura de cada sub-árvore é usada para re-equilibrar as árvores AVL. Neste exercício pretende-se que modifique a implementação apresentada na aula teórica de forma a guardar a esta informação nos nós em vez de a re-calcular todas as vezes que é usada. Comece por alterar a declaração de tipo de árvore de forma a que cada nó tenha um argumento extra para a altura:

data *Arv* *a* = *Vazia* | *No* *Int* *a* (*Arv* *a*) (*Arv* *a*)

Tenha o cuidado de modificar as funções que efectuem rotações de forma a actualizar correctamente a altura.

Verificador de tautologias

Nos exercícios seguintes pretende-se que se baseie no verificador de tautologias apresentado nas aulas teóricas.

114. Escreva uma definição da função *satisfaz* :: *Prop* → *Bool* que verifica se uma proposição é *satisfazível*, isto é, se existe uma atribuição de valores às variáveis que a torna verdadeira.
115. Escreva uma definição da função *equiv* :: *Prop* → *Prop* → *Bool* que verifica se duas proposições são *equivalentes*, isto é, tomam o mesmo valor de verdade para todas as atribuições de variáveis. Sugestão: *p*, *q* são equivalentes se e só se $p \implies q \wedge q \implies p$ for uma tautologia.
116. Modifique o verificador de tautologias acrescentando uma nova conectiva para equivalência entre proposições.
117. Escreva uma definição duma função *showProp* :: *Prop* → *String* para converter uma proposição em texto; alguns exemplos:

```
> showProp (Neg (Var 'a'))
"(~a)"
> showProp (Disj (Var 'a') (Conj (Var 'a') (Var 'b'))))
"(a || (a && b))"
> showProp (Impl (Var 'a') (Impl (Neg (Var 'a')) (Const False)))
"(a -> ((~a) -> F))"
```

118. Modifique o verificador de tautologias para imprimir a tabela de verdade duma proposição. Mais precisamente, escreva uma função *tabela* :: *Prop* → *IO ()* que imprime a tabela de verdade da proposição dada.

O Jogo da Vida

Nos exercícios seguintes pretende-se que se baseie no Jogo da Vida apresentado nas aulas teóricas.

119. Modifique a função *life* apresentada na aula teórica de forma a verificar se o tabuleiro fica vazio e, nesse caso, terminar imediatamente a simulação.
120. Modifique a função *life* apresentada na aula teórica para que no final das *n* gerações imprima o número de células vivas do tabuleiro.
121. Escreva definições das seguintes funções para transformar configurações do jogo da vida:
- (a) *trans* :: (*Int*, *Int*) → *Cells* → *Cells*, tal que *trans* (*dx*, *dy*) efectua a translação pelo vector (*dx*, *dy*) das células no tabuleiro;
 - (b) *reflh* :: *Cells* → *Cells*, que efectua a transformação de reflexão horizontal (isto é, em relação ao eixo dos *y*);
 - (c) *reflv* :: *Cells* → *Cells*, que efectua a transformação de reflexão vertical (isto é, em relação ao eixo dos *x*);
 - (d) *rot90* :: *Cells* → *Cells*, que efectua uma rotação de 90° no sentido dos ponteiros do relógio.
 - (e) Combinando estas transformações com configurações primitivas pode construir configurações complexas de forma modular. Experimente, por exemplo, simular 100 gerações de dois “gliders” em rota de colisão:
life (glider ++ trans (10,0) (reflh glider)) 100.