Cálculo de Programas
Algebra of Programming

Lic. Ciências da Computação (2º ano)
Lic./Mest.Int. em Engenharia Informática (3º ano)
UNIVERSIDADE DO MINHO

2025/26 - Ficha nr.° 1

Revisões de PF *(Functional Programming background)*

1. Complete a codificação abaixo (em Haskell) das funções $\mathsf{length} :: [\,a\,] \to \mathbb{Z}$ e $\mathsf{reverse} :: [\,a\,] \to [\,a\,]$ que conhece da disciplina de Programação Funcional (PF) e que, respectivamente, calculam o comprimento da lista de entrada e a invertem:

   Complete the code below (in Haskell) of functions $\mathsf{length} :: [\,a\,] \to \mathbb{Z}$ *and* $\mathsf{reverse} :: [\,a\,] \to [\,a\,]$ *that you know from the Functional Programming (PF) course and that, respectively, calculate the length of the input list and reverse it:*

$$\begin{array}{l} \mathsf{length}\ [\,] = \ldots \\ \mathsf{length}\ (x : xs) = \ldots \end{array}$$

$$\begin{array}{l} \mathsf{reverse}\ [\,] = \ldots \\ \mathsf{reverse}\ (x : xs) = \ldots \end{array}$$

---

2. A função $\mathsf{take} :: \mathbb{Z} \to [\,a\,] \to [\,a\,]$ é tal que $\mathsf{take}\ n\ x$ é o mais longo prefixo da lista x cujo comprimento não excede n. Complete a seguinte formulação de uma propriedade da função take:

   The function $\mathsf{take} :: \mathbb{Z} \to [\,a\,] \to [\,a\,]$ *is such that* $\mathsf{take}\ n\ x$ *is the longest prefix of list* x *whose length does not exceed* n. *Complete the following statement of a property of* take*:*

$$\mathsf{take}\ m\ (\mathsf{take}\ n\ x) = \mathsf{take}\ (m \ldots n)\ x$$

---

3. Apresente definições em Haskell das seguintes funções que estudou em PF:

   Give Haskell definitions for the following functions that you studied in the Functional Programming course:

$$\begin{array}{l} \mathsf{map} :: (a \to b) \to [\,a\,] \to [\,b\,] \\ \mathsf{filter} :: (a \to \mathbb{B}) \to [\,a\,] \to [\,a\,] \\ \mathsf{uncurry} :: (a \to b \to c) \to (a, b) \to c \\ \mathsf{curry} :: ((a, b) \to c) \to a \to b \to c \\ \mathsf{flip}\ \cdot :: (a \to b \to c) \to b \to a \to c \end{array}$$

4. A **composição** de funções define-se, em Haskell, tal como na matemática:

*Function **composition** is defined, in Haskell, just as in mathematics:*

$$(f \cdot g)\ x = f\ (g\ x) \tag{F1}$$

Calcule $(f \cdot g)\ x$ para os casos seguintes:

Evaluate $(f \cdot g)\ x$ for the following cases:

$$\left\{ \begin{array}{l} f\ x = 2 * x \\ g\ x = x + 1 \end{array} \right. \quad \left\{ \begin{array}{l} f = \mathsf{succ} \\ g\ x = 2 * x \end{array} \right. \quad \left\{ \begin{array}{l} f = \mathsf{succ} \\ g = \mathsf{length} \end{array} \right. \quad \left\{ \begin{array}{l} g\ (x, y) = x + y \\ f = \mathsf{succ} \cdot (2*) \end{array} \right.$$

Anime as composições funcionais acima num interpretador de Haskell.

Animate the above functional compositions in a Haskell interpreter.

---

5. Mostre que $(f \cdot g) \cdot h = f \cdot (g \cdot h)$, quaisquer que sejam f, g e h.

Show that the equality $(f \cdot g) \cdot h = f \cdot (g \cdot h)$ holds for all f, g and h.

---

6. A função $id :: a \rightarrow a$ é tal que $id\ x = x$. Mostre que $f \cdot id = id \cdot f = f$ qualquer que seja f.

The function $id :: a \rightarrow a$ is such that $id\ x = x$. Show that $f \cdot id = id \cdot f = f$ for all f.

---

7. Considere o seguinte problema:

Consider the following problem:

> *(...) For each **list of calls** stored in an old mobile phone (eg. numbers dialled, SMS messages, lost calls), the **store** operation should work in a way such that **(a)** the more recently a **call** is made the more accessible it is; **(b)** no number appears twice in a list; **(c)** only the most recent 10 entries in each list are stored.*

Considere ainda a seguinte proposta de resolução que usa a composição de funções, uma por cada requisito do problema:

Further consider the following solution proposal which uses function composition, one function for each requirement of the problem:

$$store\ c = \underbrace{\mathsf{take}\ 10}_{(c)} \cdot \underbrace{nub}_{(b)} \cdot \underbrace{(c:)}_{(a)} \tag{F2}$$

(a) Usando a definição (F1) tantas vezes quanto necessário, avalie as expressões

(a) Using definition (F1) as many times as needed, evaluate the expressions

$$store\ 7\ [1 \mathinner{..} 10]$$
$$store\ 11\ [1 \mathinner{..} 10]$$

(b) Suponha que alguém usou a mesma abordagem ao problema, mas enganou-se na ordem das etapas:

(b) Suppose someone used the same approach to the problem, but got the steps in the wrong order:

$$store\ c = (c:) \cdot \mathsf{take}\ 10 \cdot nub$$

Qual é o problema desta solução? Que requisitos (a,b,c) viola?

What is the problem with this solution? Which requirements (a,b,c) does it violate?

(c) E se o engano for como escreve a seguir?

(c) What if the mistake is as written below?

$$store\ c = nub \cdot (c:) \cdot \mathsf{take}\ 10$$

Conclua que a composição não é mesmo nada comutativa — a ordem entre as etapas de uma solução composicional é importante!

Conclude that composition is not commutative at all — the order between the steps of a compositional solution is important!

---

8. Voltando a agora à definição *certa* (F2), suponha que submete ao seu interpretador de Haskell a expressão:

Returning to definition (F2), suppose you submit the following expression to your Haskell interpreter:

store `"Mary"` `["Manuel","Tia Irene","Mary","Augusto"]`

Que espera do resultado? Vai dar erro? Tem que mexer em (F2) para funcinar? Que propriedade da linguagem é evidenciada neste exemplo?

What is the outcome you expect? Will it be an error? Do I need to change (F2) for the above to work? What property of the Haskell programming language is made evident in this example?