Cálculo de Programas
Resolução - Ficha 01

Eduardo Freitas Fernandes

2025

Exercício 1

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

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

Exercício 2

```haskell
take m (take n x) = take (m `min` n) x
```

Exercício 3

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

filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter f (x:xs) = if f x then x : filter f xs else filter f xs

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

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

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

Exercício 4

$$\begin{cases} f\ x = 2 * x \\ g\ x = x + 1 \end{cases} \quad (f \cdot g)\ x \overset{(\mathrm{F1})}{=} f(g\ x) = 2 * (x + 1)$$

```
ghci> f x = 2 * x
ghci> g x = x + 1
ghci> (f . g) 5
12
```

$$\begin{cases} f\ x = succ \\ g\ x = 2 * x \end{cases} \quad (f \cdot g)\ x \overset{(\mathrm{F1})}{=} f(g\ x) = succ(2 * x)$$

```
ghci> f = succ
ghci> g x = 2 * x
ghci> (f . g) 5
11
```

$$\begin{cases} f\ x = succ \\ g\ x = length \end{cases} \quad (f \cdot g)\ x \overset{(\mathrm{F1})}{=} f(g\ x) = succ(length)$$

```
ghci> f = succ
ghci> g = length
ghci> (f . g) [1..7]
8
```

$$\begin{cases} g\ (x, y) = x + y \\ f = succ \cdot (2*) \end{cases} \quad (f \cdot g)\ x \overset{(\mathrm{F1})}{=} f(g\ x) = succ(2 * (x + y))$$

```
ghci> g (x,y) = x+y
ghci> f = succ . (2*)
ghci> (f . g) (3,5)
17
```

Exercício 5

$$\begin{aligned} (f \cdot g) \cdot h\ x &\overset{(\mathrm{F1})}{=} (f \cdot g)(h\ x) \\ &\overset{(\mathrm{F1})}{=} f(g(h\ x)) \\ &\overset{(\mathrm{F1})}{=} f((g \cdot h)\ x) \\ &\overset{(\mathrm{F1})}{=} f \cdot (g \cdot h)\ x \end{aligned}$$

Exercício 6

Iremos assumir a seguite definição da função identidade:

```
id :: a -> a
id x = x
```

$$f \cdot id\ x \overset{\text{(F1)}}{=} f(id\ x) \overset{\text{def. id}}{=} f\ x$$

$$id \cdot f\ x \overset{\text{(F1)}}{=} id(f\ x) \overset{\text{def. id}}{=} f\ x$$

Exercício 7

a)

$$\begin{aligned}
store\ 7\ [1..10] &\overset{\text{(F2)}}{=} take\ 10 \cdot nub \cdot 7 : [1..10] \\
&\overset{\text{(F1)}}{=} take\ 10\ (nub\ (7 : [1..10])) \\
&\overset{\text{def. :}}{=} take\ 10\ (nub\ [7,1,2,3,4,5,6,7,8,9,10]) \\
&\overset{\text{def. nub}}{=} take\ 10\ [7,1,2,3,4,5,6,8,9,10] \\
&\overset{\text{def. take 10}}{=} [7,1,2,3,4,5,6,8,9,10] \\
store\ 11\ [1..10] &\overset{\text{(F2)}}{=} take\ 10 \cdot nub \cdot 11 : [1..10] \\
&\overset{\text{(F1)}}{=} take\ 10\ (nub\ (11 : [1..10])) \\
&\overset{\text{def. :}}{=} take\ 10\ (nub\ [11,1,2,3,4,5,6,7,8,9,10]) \\
&\overset{\text{def. nub}}{=} take\ 10\ [11,1,2,3,4,5,6,7,8,9,10] \\
&\overset{\text{def. take 10}}{=} [11,1,2,3,4,5,6,7,8,9]
\end{aligned}$$

b) o requisito (a) é cumprido mas os restantes não, pois a primeira operação a ser feita é a remoção de duplicados. Esta operação deve ser feita após adicionar `c`. O requisito (b) não é cumprido, pois primeiro seleciona-se os 10 primeiros elementos e depois adiciona-se `c`, logo a lista final terá no máximo 11 elementos, violando o requisito (c).

c) o requisito (c) é violado, pois primeiro retira-se os primeiros 10 elementos e adiciona-se `c`, logo a lista final terá no máximo 11 elementos (caso a função `nub` não efetue mudanças na lista).

Exercício 8

O resultado será `["Mary","Manuel","Tia Irene","Augusto"]`. Neste exemplo é evidenciado o facto das funções en Haskell só receberem um argumento. Ao analisar a composição de funções em (F2):

- `(c:) :: [a] -> [a]`
- `nub :: [a] -> [a]`
- `take 10 :: [a] -> [a]`

verificamos que a função `store` não recebe dois argumentos, recebe um argumento `c` e devolve uma função do tipo `store c :: [a] -> [a]`. Isto acontece porque Haskell aplica **currying** automaticamente.