Programação Funcional — Exame 619r

27 de Junho de 2019 — DCC/FCUP

Nome: ____________________

Nº mecanográfico: ____________________

- **Este exame contém 7 questões em 4 páginas e tem uma duração de 2h(+30m).**
- **Responda às questões no espaço marcado no enunciado.**
- **Pode usar funções auxiliares e/ou do prelúdio-padrão de Haskell.**
- **Nas questões 2 a 7, indique sempre o tipo da função definida.**

1. (30%) Responda a cada uma das seguintes questões, indicando **apenas** o resultado de cada expressão.

(a) `[]:[]:[] ++ []:[]:[]` = `[[],[],[],[]]`

(b) `length ([]:[]:[] ++ []:[]:[])` = `4`

(c) `map (`div` 4) [1..10]` = `[0,0,0,1,1,1,1,2,2,2]`

(d) `takeWhile (<100) (iterate (3*) 1)` = `[1,3,9,27,81]`

(e) `(filter even.(!!2)) [[3,5,8],[4],[7,4,1,8],[9,11]]` = `[4,8]`

(f) `take 10 [ 2*x*y | x<-[1..], y <-[1..]]` = `[2,4,6,8,10,12,14,16,18,20]`

(g) Defina a lista infinita `[0,1,4,7,8,31,12,127,16,511,...]` em compreensão:

```
[if even x then 2*x else 2^x-1 | x <-[0..]]
```

(h) Considere a seguinte definição em Haskell:

```
f p [] = True
f p (x:xs) = p x && f p xs
```

A avaliação da expressão `f (>3) [3,4,5]` tem como resultado: `False`

(i) Indique um tipo admissível para `[even,(==2),(>1)]`: `[Int -> Bool]`

(j) Indique o tipo mais geral de `length.(filter (>'a'))`: `[Char] -> Int`

(k) Considere as seguintes definições em Haskell:

```
data ???

f :: Arv -> Int
f (No esq a dir) | dir == Vazia = a
                 | otherwise = f dir
```

O que falta em `???` para que a função `f` esteja bem definida?

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

(l) Indique o tipo mais geral de `flip f x y = f y x`: `(a -> b -> c) -> b -> a -> c`

2. (15%) Considere a função `imparDiv3`, que dada uma lista de inteiros, verifica que cada inteiro na lista que é divisível por 3 é ímpar. Por exemplo `imparDiv3 [1,15,153,83,64,9] = True` e `imparDiv3 [1,12,153,83,9] = False`.

(a) Defina a função `imparDiv3` usando listas em compreensão e funções do prelúdio, mas não **recursão**.

(b) Defina a função `imparDiv3` usando funções de ordem-superior (`map, filter, foldr`), mas não **recursão** ou **listas em compreensão**.

Resolução:

(a)

```
imparDiv3:: [Int] -> Bool
imparDiv3 xs = and [ odd x | x <- xs, x `mod` 3 == 0]
```

(b)

```
imparDiv3:: [Int] -> Bool
imparDiv3 = (all odd) . (filter (\x -> x `mod` 3 == 0))
```

3. (10%) Considere o seguinte tipo `Rel a`, definido como `type Rel a = [(a,[a])]`, para representar relações binárias, como uma lista de adjacências (*Nota: a ordem dos elementos na lista de nós adjacentes não é importante, mas não devem aparecer elementos repetidos na lista de nós adjacentes, nem devem existir dois pares na relação com o mesmo primeiro elemento*). Por exemplo a relação $R = \{(1,1),(1,3),(2,2),(3,2),(3,3)\}$ é representada por `r = [(1,[1,3]),(2,[2]),(3,[2,3])]`. Defina uma função `composta`, que dadas duas relações R_1 e R_2, representadas da forma descrita acima, calcule a relação composta das duas. (Nota: $R_1 \cdot R_2 = \{(x,z) \mid (x,y) \in R_1 \wedge (y,z) \in R_2\}$.) Por exemplo `composta r r = [(1,[1,3,2]),(2,[2]),(3,[2,3])]`. **Sugestão:** poderá utilizar a função `nub` do módulo `Data.List` para remover elementos repetidos.

Resolução:

```
composta :: Eq a => Rel a -> Rel a -> Rel a
composta xs ys = [(x, nub (concat [l2 | (y,l2) <- ys, elem y l1])) | (x,l1) <- xs]
```

4. (5%) Considere a seguinte série definida por recorrência da seguinte forma: $a_1 = 1$, $a_n = 2 * a_{n-1} + n + 1$, $n > 1$. Defina em Haskell a lista infinita `[1,5,14,33,72,151,310,629,...]`, contendo todos os termos da série. *Nota: serão valorizadas soluções mais eficientes.*

Resolução:

```
serie :: [Int]
serie = 1:[ 2*x+y+1 | (x,y) <- zip serie [2..]]
```

5. (15%) Considere listas em que cada valor aparece sequencialmente duplicado. Por exemplo `[1,1,2,2,3,3]` ou `[True,True,False,False]`.

(a) Defina recursivamente a função `duplicada`, que dada uma lista `xs`, verifica que a lista satisfaz a condição acima. Por exemplo `duplicada [1,1,2,2,3,3] = True` e `duplicada [True,False,False] = False`.

(b) Usando ordem superior e/ou listas em compreensão, defina a função `duplica`, que dada uma lista `xs`, constrói uma nova lista de valores duplicados. Por exemplo, `duplica [1,2,3,4] = [1,1,2,2,3,3,4,4]` e `duplica [True,False] = [True,True,False,False]`.

Resolução:

(a)

```
duplicada:: Eq a => [a] -> Bool
duplicada [] = True
duplicada [_] = False
duplicada (x:y:xs) = x == y && duplicada xs
```

(b)

```
duplica:: Eq a => [a] -> Bool
duplica xs = concat [[x,x] | x <- xs]
```

ou

```
duplica:: Eq a => [a] -> Bool
duplica xs = [x | x <- xs, _ <- [1,2]]
```

ou

```
duplica:: Eq a => [a] -> Bool
duplica xs =  foldr (\a b -> a:a:b) [] xs
```

6. (15%) Considere a seguinte declaração de tipo para árvores binárias com valores nas folhas:
`data Arv a = Folha a | No (Arv a) (Arv a)`

(a) Defina a função `emOrdem`, que dada uma árvore do tipo `Arv a`, retorna a lista de elementos na árvore, seguindo a ordem da esquerda para a direita.

(b) Recorde a função de ordem-superior `any:: (a->Bool) -> [a] -> Bool` definida para listas. Defina uma função `anyArv`, que se comporte como a função `any`, mas opere sobre árvores do tipo `Arv a`.

Resolução:

(a)

```
emOrdem :: Arv a -> [a]
emOrdem (Folha v) = [v]
emOrdem (No esq dir) = emOrdem esq ++ emOrdem dir
```

(b)

```
anyArv:: (a->Bool) -> Arv a -> Bool
anyArv p (Folha v) = p v
anyArv p (No esq dir) = anyArv p esq || anyArv p dir
```

7. (10%) Responda **(apenas)** a uma das seguintes alíneas, usando indução matemática.
Nota: pode utilizar qualquer propriedade que tenha sido demostrada nas aulas. Pode ainda usar sem demonstrar as seguintes propriedades sobre listas: `any (xs++ys) = any xs || any ys` *e* `foldr f v (xs ++ ys) = foldr f (foldr f v ys) xs`.

(a) Considerando as funções `emOrdem` e `anyArv` definidas na questão anterior e a função `any` do prelúdio, mostre que para qualquer árvore `t` e predicado `p`, `anyArv p t = any p (emOrdem t)` .

(b) Considerando as definições das funções `foldr` e `concat` dadas nas aulas, assim como a função `flip` do prelúdio, definida como `flip f x y = f y x`, mostre que para quaisquer `f` e `v` e `xs`: `foldr f v (concat xs) = foldr (flip (foldr f)) v xs`.

Resolução:

(a) Por inducão sobre `t`.

- Caso base $\mathtt{t} = \mathtt{Folha\ v}$:

$$\begin{array}{rcl} \mathtt{anyArv\ p\ (Folha\ v)} & = & \mathtt{p\ v} \\ & = & \mathtt{p\ v} \mid\mid \mathtt{False} = \mathtt{any\ p}\ [\mathtt{x}] = \mathtt{any\ p\ (emOrdem\ (Folha\ v))} \end{array}$$

- Caso indutivo $\mathtt{t} = \mathtt{No\ esq\ dir}$:
  - Hipótese:

```
anyArv p esq = any p (emOrdem esq)
anyArv p dir = any p (emOrdem dir)
```

  - Tese: $\mathtt{anyArv\ p\ (No\ esq\ dir)} = \mathtt{any\ p\ (emOrdem\ (No\ esq\ dir))}$.

$$\begin{array}{rcl} \mathtt{anyArv\ p\ (No\ esq\ dir)} & = & \mathtt{anyArv\ p\ esq} \mid\mid \mathtt{anyArv\ p\ dir} \\ & = & \mathtt{any\ p\ (emOrdem\ esq)} \mid\mid \mathtt{any\ p\ (emOrdem\ dir)} \\ & = & \mathtt{any\ p\ (emOrdem\ esq\ ++\ emOrdem\ dir)} \\ & = & \mathtt{any\ p\ (emOrdem\ (No\ esq\ dir))} \end{array}$$

(b) Por inducão sobre `xs`.

- Caso base $\texttt{xs} = [\,]$:

$$\begin{aligned}\texttt{foldr f v (concat } [\,]) &= \texttt{foldr f v } [\,] \\ &= \texttt{v} = \texttt{foldr (flip (foldr f)) v } [\,]\end{aligned}$$

- Caso indutivo $\texttt{xs} = (\texttt{y} : \texttt{ys})$:
  - Hipótese: $\texttt{foldr f v (concat ys)} = \texttt{foldr (flip (foldr f)) v ys}$
  - Tese: $\texttt{foldr f v (concat (y : ys))} = \texttt{foldr (flip (foldr f)) v (y : ys)}$.

$$\begin{aligned}\texttt{foldr f v (concat (y : ys)} &= \texttt{foldr f v (y} \mathbin{++} \texttt{(concat ys))} \\ &= \texttt{foldr f (foldr f v (concat ys)) y} \\ &= \texttt{foldr f (foldr (flip (foldr f)) v ys) y} \\ &= \texttt{flip (foldr f) y (foldr (flip (foldr f)) v ys)} \\ &= \texttt{foldr (flip (foldr f)) v (y : ys)}\end{aligned}$$