Programação Funcional, 2018.1 **Prova 2.1**

(Turma T56 do Thanos) (points: 126; bonus: $0^\flat$; time: 80')

Nome:

02/07/2018

Regras:

I. Não vires esta página antes do começo da prova.

II. Nenhuma consulta de qualquer forma.

III. Nenhum aparelho ligado (por exemplo: celular, tablet, notebook, *etc.*).[1]

IV. Nenhuma comunicação de qualquer forma e para qualquer motivo.

V. $\forall x\big(\text{Colar}(x) \to \neg\text{Passar}(x, \text{FUN})\big)$.

VI. Use caneta para tuas respostas.

VII. Responda dentro das caixas indicadas.

VIII. Escreva teu nome em *cada* folha de rascunho extra *antes de usá-la*.

IX. Entregue *todas* as folhas de rascunho extra, juntas com tua prova.

X. Nenhuma prova será aceita depois do fim do tempo!

XI. Os pontos bônus são considerados apenas para quem consiga passar sem.[2]

XII. Responda em até 3 dos problemas.[3]

Boas provas!

[1] Ou seja, *desligue antes* da prova.

[2] Por exemplo, 25 pontos bônus podem aumentar uma nota de 5,2 para 7,7 ou de 9,2 para 10,0, mas de 4,9 nem para 7,4 nem para 5,0. A 4,9 ficaria 4,9 mesmo.

[3] Provas com respostas em mais que três problemas não serão corrigidas (tirarão 0 pontos).

(32) **A**

Defina os típos de dados:

```
Nat    :: *          List   :: * -> *         Maybe  :: * -> *
```

e as funções

```
(+),(*) :: Nat -> Nat -> Nat              filter :: (a -> Bool) -> List a -> List a
len     :: List a -> Nat                  head   :: List a -> Maybe a
take    :: Nat -> List a -> List a        (++)   :: List a -> List a -> List a
map     :: (a -> b) -> List a -> List b   rev    :: List a -> List a
```

Definições.

(42) # B

Prove **exatamente um** dos dois teoremas.

(42) **B1.** `(+)` é comutativa para todo número finito.

(28) **B2.** `(++)` é associativa para toda lista finita.

Prova da _____.

(42) **F**

Defina a

`foldr :: (a -> b -> b) -> b -> List a -> b`

e depois a

`why :: List String -> (String, Maybe (List String), Int)`

como um fold. Ela retorna: o string feito pelas primeiras letras de todo string da sua entrada; se aparecem palindromos, uma lista de todos eles; e o número dos palindromos. Exemplo de uso:

```
why [ "hello"
    , "mom"
    , "do you see"
    , "bob"
    , "over there"
    ]
    = ("hmdbo", Just ["mom", "bob"], 2)
why [ "be", "right", "back" ] = ("brb", Nothing, 0)
```

Definições.

(36) # M

Prove **exatamente uma** das leis de `map`:

(24) **M1.** `map id = id`.

(36) **M2.** `map (g . h) = map g . map h`.

Prova de _____.

(42) **L**

Considere o tipo de arvores seguinte

```
data Tree  =  Leaf Int | Node Tree Tree
```

e a função `flatten` definida assim:

```
flatten :: Tree -> [Int]
flatten (Leaf n)   = [n]
flatten (Node l r) = flatten l ++ flatten r
```

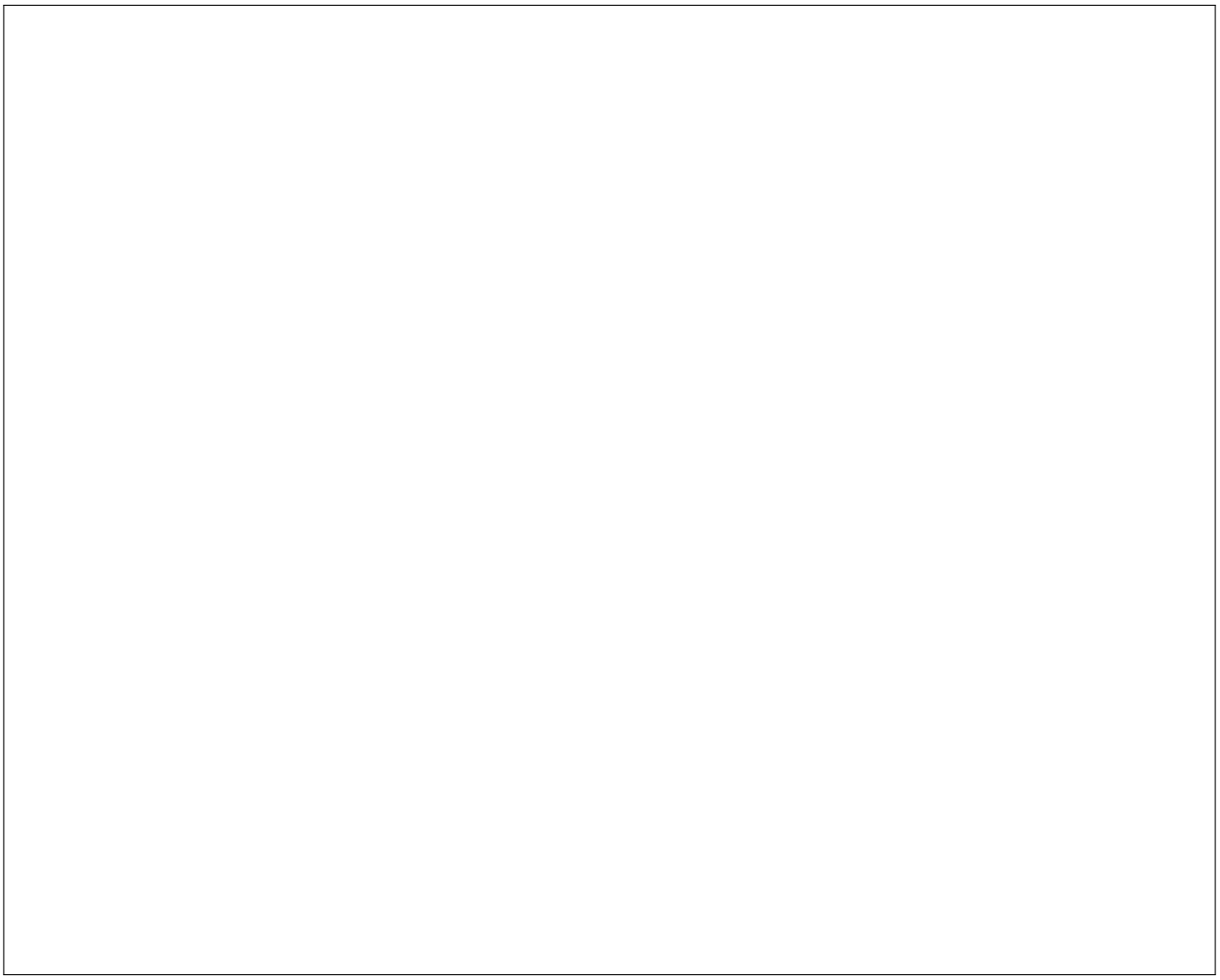
Mostre como definir uma versão de `flatten` mais eficiente, usando indução para definir uma função auxiliar `flatten' :: Tree -> [Int] -> [Int]` que satisfaz

```
flatten' t ns = flatten t ++ ns
```

mas (obviamente) não usa o `flatten` na sua definição. Tua definição deve ter a forma:

```
flatten' (Leaf n)   ns =  ...
flatten' (Node l r) ns =  ...
```

Prova/Definição.

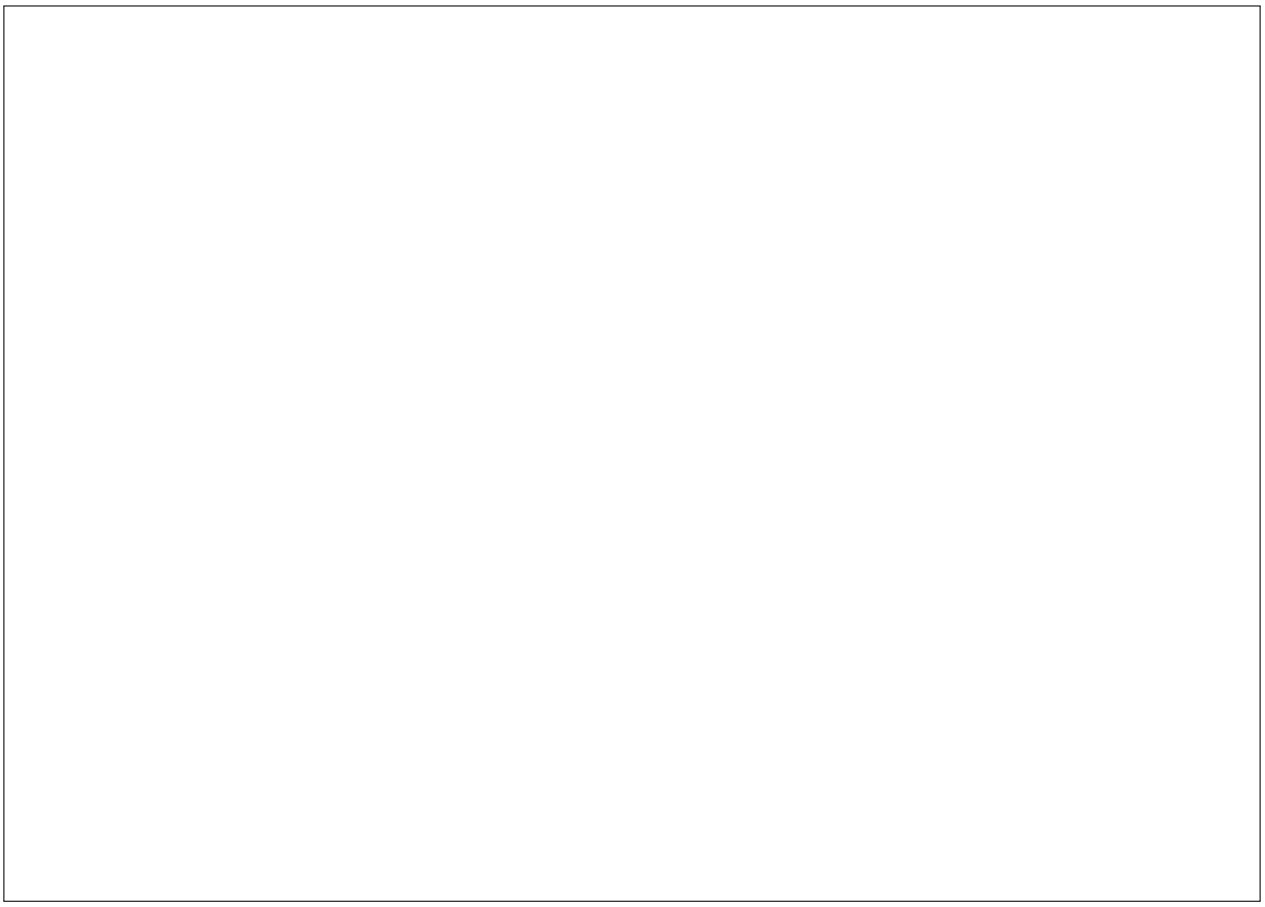


(24) # R

Defina recursivamente **exatamente duas** das funções seguintes:

```
inits   :: [a] -> [[a]]
tails   :: [a] -> [[a]]
subseqs :: [a] -> [[a]]
```

Definições.



(18) **I**

Definimos as

```
one x  = [x]
none x = []
```

Completa as equações (sem justificação)

```
none  . f     =

map f . none  =

map f . one   =
```

Lembre-se que

```
pair :: (a -> b,a -> c) -> a -> (b,c)
pair (f,g) x = (f x, g x)
```

Completa as equações (sem justificação)

```
fst . pair (f,g) =

snd . pair (f,g) =

pair (f,g) . h   =
```

Só isso mesmo.

Rascunho

Rascunho