Princípios de Programação

Trabalho Segundo

Universidade de Lisboa
Faculdade de Ciências
Departamento de Informática
Licenciatura em Engenharia Informática

2020/2021

Neste trabalho segundo vamos desenvolver três pequenas aplicações.

A. Diferencas consecutivas
1. Escreva uma função recursiva polimórfica que, dada uma lista, devolva uma lista com os pares de elementos consecutivos, pela ordem em que os elementos aparecem na lista. Por exemplo

```
> paresConsecutivos "ola"
[('o','l'),('l','a')]
> paresConsecutivos [1..7]
[(1,2),(2,3),(3,4),(4,5),(5,6),(6,7)]
> paresConsecutivos [True]
[]
> paresConsecutivos []
[]
```

2. Escreva uma função recursiva que, dada uma lista de inteiros, devolve a lista das diferenças entre cada elemento e o anterior, pela ordem em que os elementos aparecem na lista. Por exemplo

```
> diferencasConsecutivas [1, 4, 8, 2, -4]
[3,4,-6,-6]
> diferencasConsecutivas [3, 6 .. 20]
[3,3,3,3,3]
> diferencasConsecutivas [27]
[]
> diferencasConsecutivas []
[]
```

B. Jogadores de futebol Um selecionador de futebol pretende escolher um conjunto de jogadores que não sejam claramente piores que qualquer outro jogador. Considere que cada jogador é representado por um triplo: nome, pontuação de ataque (inteiro entre 0 e 10) e pontuação de defesa (inteiro entre 0 e 10). Por exemplo

```
jogador :: (String, Int, Int)
jogador = ("Ronaldo", 10, 3)
```

Um jogador é claramente pior que outro se tiver ambas as pontuação de ataque e de defesa inferiores. Por exemplo, um jogador que tenha ataque de 7 e defesa de 2 não é pior que um jogador com 2 de ataque e 5 de defesa. Já um jogador com 1 em ambos os critérios é claramente pior e não deveria ser selecionado.

1. Escreva uma função recursiva que, dado um jogador e uma lista de jogadores, indica se esse jogador é claramente pior que algum outro jogador na lista. Por exemplo

```
> claramentePior ("Marco", 2, 5) [("Ronaldo", 10, 3),
    ("Pedro", 3, 6)]
True
> claramentePior ("Bruno", 5, 4) [("Ronaldo", 10, 3),
    ("Pedro", 3, 6)]
False
```

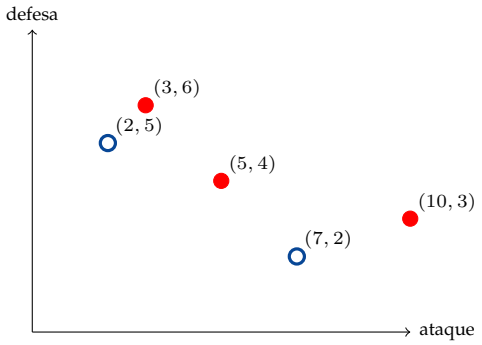

Figura 1: Jogadores de uma equipa. Os círculos vermelhos representam os jogadores que não são claramente piores que nenhum outro jogador.

2. Escreva uma função que recebe uma lista de jogadores e devolve os jogadores que não são claramente piores que nenhum outro jogador (veja a Figura 1). Por exemplo

```
> filtroJogadores [("Ronaldo", 10, 3), ("Marco", 2, 5),
    ("Pedro", 3, 6), ("Bruno", 5, 4), ("Zé", 7, 2)]
[("Ronaldo", 10, 3), ("Pedro", 3, 6), ("Bruno", 5, 4)]
```

C. Preencher o buraco Um problema de Sudoku pode ser descrito por uma matriz quadrada de dimensão 9 onde cada entrada na tabela contém um número inteiro entre 1 e 9. Para descrever valores por preencher na tabela usamos o número 0. Eis um Sudoku onde apenas o canto inferior direito se encontra por preencher.

```
s :: [[Int]]
s = [ [3, 6, 4, 8, 7, 1, 2, 9, 5]
    , [7, 5, 2, 9, 3, 6, 1, 8, 4]
    , [8, 1, 9, 2, 5, 4, 7, 3, 6]
    , [5, 9, 6, 7, 1, 3, 4, 2, 8]
    , [4, 3, 1, 5, 8, 2, 6, 7, 9]
    , [2, 7, 8, 4, 6, 9, 3, 5, 1]
    , [6, 4, 5, 3, 2, 8, 9, 1, 7]
    , [9, 8, 3, 1, 4, 7, 5, 6, 2]
    , [1, 2, 7, 6, 9, 5, 8, 4, 0]
    ]
```

No exemplo acima, se trocarmos o `0` por `3` obtemos um Sudoku *válido*, isto é, um Sudoku onde cada algarismo de 1 a 9 aparece uma única vez em cada linha, em cada coluna e em cada um dos nove sub-quadrados de lado três. Escreva a função `preencherVazio` que dado um problema de Sudoku (uma lista de listas de inteiros) devolve o número que deverá substituir o `0` de modo a completar o Sudoku. Exemplo:

```
> preencherVazio s
3
```

Assuma que

- O Sudoku tem exatamente uma casa vazia, isto é, a matriz tem exatamente um número `0` e
- O Sudoku tem solução, isto é, existe um número que, substituindo o `0`, torna a matriz num Sodoku válido.

Notas

1. Os trabalhos serão avaliados automaticamente. Respeite os nomes e os tipos de todas as funções enunciadas acima.
2. Cada função (ou expressão) que escrever deverá vir sempre acompanhada de uma assinatura. Isto é válido para as funções enunciadas acima bem como para outras funções ajudantes que decidir implementar.
3. Para resolver estes problemas deve utilizar *apenas* a matéria dos cinco primeiros capítulos do livro (até "Recursion" inclusivé). Pode usar qualquer função constante no **`Prelude`**. Não pode em caso algum utilizar funções de ordem superior (capítulo 6).

4. Lembre-se que as boas práticas de programação Haskell apontam para a utilização de várias funções simples em lugar de uma função única mas complicada.

Entrega. Este é um trabalho de resolução individual. Os trabalhos devem ser entregues no Moodle até às 23:55 do dia 19 de outubro de 2020.

Plágio. Os trabalhos de todos os alunos serão comparados por uma aplicação computacional. Relembramos aqui um excerto da sinopse: "Alunos detetados em situação de fraude ou plágio, plagiadores e plagiados, ficam reprovados à disciplina (sem prejuízo de ser acionado processo disciplinar concomitante)".