Programação Funcional

1º Ano – LCC/MIEFis/MIEI

Questões 1ª Parte

1. Apresente uma definição recursiva da função (pré-definida) `enumFromTo :: Int -> Int -> [Int]` que constrói a lista dos números inteiros compreendidos entre dois limites.

   Por exemplo, `enumFromTo 1 5` corresponde à lista `[1,2,3,4,5]`

2. Apresente uma definição recursiva da função (pré-definida) `enumFromThenTo :: Int -> Int -> Int -> [Int]` que constrói a lista dos números inteiros compreendidos entre dois limites e espaçados de um valor constante.

   Por exemplo, `enumFromThenTo 1 3 10` corresponde à lista `[1,3,5,7,9]`.

3. Apresente uma definição recursiva da função (pré-definida) `(++) :: [a] -> [a] -> [a]` que concatena duas listas.

   Por exemplo, `(++) [1,2,3] [10,20,30]` corresponde à lista `[1,2,3,10,20,30]`.

4. Apresente uma definição recursiva da função (pré-definida) `(!!) :: [a] -> Int -> a` que dada uma lista e um inteiro, calcula o elemento da lista que se encontra nessa posição (assume-se que o primeiro elemento se encontra na posição `0`).

   Por exemplo, `(!!) [10,20,30] 1` corresponde a `20`.

   Ignore os casos em que a função não se encontra definida (i.e., em que a posição fornecida não corresponde a nenhuma posição válida da lista).

5. Apresente uma definição recursiva da função (pré-definida) `reverse :: [a] -> [a]` que dada uma lista calcula uma lista com os elementos dessa lista pela ordem inversa.

   Por exemplo, `reverse [10,20,30]` corresponde a `[30,20,10]`.

6. Apresente uma definição recursiva da função (pré-definida) `take :: Int -> [a] -> [a]` que dado um inteiro `n` e uma lista `l` calcula a lista com os (no máximo) `n` primeiros elementos de `l`.

   A lista resultado só terá menos de que `n` elementos se a lista `l` tiver menos do que `n` elementos. Nesse caso a lista calculada é igual à lista fornecida.

   Por exemplo, `take 2 [10,20,30]` corresponde a `[10,20]`.

7. Apresente uma definição recursiva da função (pré-definida) `drop :: Int -> [a] -> [a]` que dado um inteiro `n` e uma lista `l` calcula a lista sem os (no máximo) `n` primeiros elementos de `l`.

   Se a lista fornecida tiver `n` elementos ou menos, a lista resultante será vazia.

   Por exemplo, `drop 2 [10,20,30]` corresponde a `[30]`.

8. Apresente uma definição recursiva da função (pré-definida) `zip :: [a] -> [b] -> [(a,b)]` constói uma lista de pares a partir de duas listas.

   Por exemplo, `zip [1,2,3] [10,20,30,40]` corresponde a `[(1,10),(2,20),(3,30)]`.

9. Apresente uma definição recursiva da função (pré-definida) `elem :: Eq a => a -> [a] -> Bool` que testa se um elemento ocorre numa lista.

   Por exemplo, `elem 20 [10,20,30]` corresponde a `True` enquanto que `elem 2 [10,20,30]` corresponde a `False`.

10. Apresente uma definição recursiva da função (pré-definida) `replicate :: Int -> a -> [a]` que dado um inteiro `n` e um elemento `x` constói uma lista com `n` elementos, todos iguais a `x`.

    Por exemplo, `replicate 3 10` corresponde a `[10,10,10]`.

11. Apresente uma definição recursiva da função (pré-definida) `intersperse :: a -> [a] -> [a]` que dado um elemento e uma lista, constrói uma lista em que o elemento fornecido é *intercalado* entre os elementos da lista fornecida.

    Por exemplo, `intersperce 1 [10,20,30]` corresponde a `[10,1,20,1,30]`.

12. Apresente uma definição recursiva da função (pré-definida) `group :: Eq a => [a] -> [[a]]` que agrupa elementos iguais e consecutivos de uma lista.

    Por exemplo, `group [1,2,2,3,4,4,4,5,4]` corresponde a `[[1],[2,2],[3],[4,4,4],[5],[4]]`.

13. Apresente uma definição recursiva da função (pré-definida) `concat :: [[a]] -> [a]` que concatena as listas de uma lista.

    Por exemplo, `concat [[1],[2,2],[3],[4,4,4],[5],[4]]` corresponde a `[1,2,2,3,4,4,4,5,4]`.

14. Apresente uma definição recursiva da função (pré-definida) `inits :: [a] -> [[a]]` que calcula a lista dos prefixos de uma lista.

    Por exemplo, `inits [11,21,13]` corresponde a `[[],[11],[11,21],[11,21,13]]`.

15. Apresente uma definição recursiva da função (pré-definida) `tails :: [a] -> [[a]]` que calcula a lista dos sufixos de uma lista.

    Por exemplo, `tails [1,2,3]` corresponde a `[[1,2,3],[2,3],[3],[]]`.

16. Apresente uma definição recursiva da função (pré-definida) `isPrefixOf :: Eq a => [a] -> [a] -> Bool` que testa se uma lista é prefixo de outra.

    Por exemplo, `isPrefixOf [10,20] [10,20,30]` corresponde a `True` enquanto que `isPrefixOf [10,30] [10,20,30]` corresponde a `False`.

17. Apresente uma definição recursiva da função (pré-definida) `isSuffixOf :: Eq a => [a] -> [a] -> Bool` que testa se uma lista é sufixo de outra.

    Por exemplo, `isSuffixOf [20,30] [10,20,30]` corresponde a `True` enquanto que `isSuffixOf [10,30] [10,20,30]` corresponde a `False`.

18. Apresente uma definição recursiva da função (pré-definida) `isSubsequenceOf :: Eq a => [a] -> [a] -> Bool` que testa se os elementos de uma lista ocorrem noutra pela mesma ordem relativa.

    Por exemplo, `isSubsequenceOf [20,40] [10,20,30,40]` corresponde a `True` enquanto que `isSubsequenceOf [40,20] [10,20,30,40]` corresponde a `False`.

19. Apresente uma definição recursiva da função (pré-definida) `elemIndices :: Eq a => a -> [a] -> [Int]` que calcula a lista de posições em que um dado elemento ocorre numa lista.

    Por exemplo, `elemIndices 3 [1,2,3,4,3,2,3,4,5]` corresponde a `[2,4,6]`.

20. Apresente uma definição recursiva da função (pré-definida) `nub :: Eq a => [a] -> [a]` que calcula uma lista com os mesmos elementos da recebida, sem repetições.

    Por exemplo, `nub [1,2,1,2,3,1,2]` corresponde a `[1,2,3]`.

21. Apresente uma definição recursiva da função (pré-definida) `delete :: Eq a => a -> [a] -> [a]` que retorna a lista resultante de remover (a primeira ocorrência de) um dado elemento de uma lista.

    Por exemplo, `delete 2 [1,2,1,2,3,1,2]` corresponde a `[1,1,2,3,1,2]`. Se não existir nenhuma ocorrência a função deverá retornar a lista recebida.

22. Apresente uma definição recursiva da função (pré-definida) `(\\):: Eq a => [a] -> [a] -> [a]` que retorna a lista resultante de remover (as primeiras ocorrências) dos elementos da segunda lista da primeira.

    Por exemplo, `(\\)[1,2,3,4,5,1] [1,5]` corresponde a `[2,3,4,1]`.

23. Apresente uma definição recursiva da função (pré-definida) `union :: Eq a => [a] -> [a] -> [a]` que retorna a lista resultante de acrescentar à primeira lista os elementos da segunda que não ocorrem na primeira.

    Por exemplo, `union [1,1,2,3,4] [1,5]` corresponde a `[1,1,2,3,4,5]`.

24. Apresente uma definição recursiva da função (pré-definida) `intersect :: Eq a => [a] -> [a] -> [a]` que retorna a lista resultante de remover da primeira lista os elementos que não pertencem à segunda.

    Por exemplo, `intersect [1,1,2,3,4] [1,3,5]` corresponde a `[1,1,3]`.

25. Apresente uma definição recursiva da função (pré-definida) `insert :: Ord a => a -> [a] -> [a]` que dado um elemento e uma lista ordenada retorna a lista resultante de inserir ordenadamente esse elemento na lista.

    Por exemplo, `insert 25 [1,20,30,40]` corresponde a `[1,20,25,30,40]`.

26. Apresente uma definição recursiva da função (pré-definida) `unwords :: [String] -> String` que junta todas as strings da lista numa só, separando-as por um espaço.

    Por exemplo, `unwords ["Programacao", "Funcional"]` corresponde a `"Programacao Funcional"`.

27. Apresente uma definição recursiva da função (pré-definida) `unlines :: [String] -> String` que junta todas as strings da lista numa só, separando-as pelo caracter `'\n'`.

    Por exemplo, `unlines ["Prog", "Func"]` corresponde a `"Prog\nFunc\n"`.

28. Apresente uma definição recursiva da função `pMaior :: Ord a => [a] -> Int` que dada uma lista não vazia, retorna a posição onde se encontra o maior elemento da lista. As posições da lista começam em `0`, i.e., a função deverá retornar `0` se o primeiro elemento da lista for o maior.

29. Apresente uma definição recursiva da função `temRepetidos :: Eq a => [a] -> Bool` que testa se uma lista tem elementos repetidos.

    Por exemplo, `temRepetidos [11,21,31,21]` corresponde a `True` enquanto que `temRepetidos [11,2,31,4]` corresponde a `False`.

30. Apresente uma definição recursiva da função `algarismos :: [Char] -> [Char]` que determina a lista dos algarismos de uma dada lista de caracteres.

    Por exemplo, `algarismos "123xp5"` corresponde a `"1235"`.

31. Apresente uma definição recursiva da função `posImpares :: [a] -> [a]` que determina os elementos de uma lista que ocorrem em posições ímpares. Considere que o primeiro elemento da lista ocorre na posição `0` e por isso par.

    Por exemplo, `posImpares [10,11,7,5]` corresponde a `[11,5]`.

32. Apresente uma definição recursiva da função `posPares :: [a] -> [a]` que determina os elementos de uma lista que ocorrem em posições pares. Considere que o primeiro elemento da lista ocorre na posição `0` e por isso par.

    Por exemplo, `posPares [10,11,7,5]` corresponde a `[10,7]`.

33. Apresente uma definição recursiva da função `isSorted :: Ord a => [a] -> Bool` que testa se uma lista está ordenada por ordem crescente.

    Por exemplo, `isSorted [1,2,2,3,4,5]` corresponde a `True`, enquanto que `isSorted [1,2,4,3,4,5]` corresponde a `False`.

34. Apresente uma definição recursiva da função `iSort :: Ord a => [a] -> [a]` que calcula o resultado de ordenar uma lista. Assuma, se precisar, que existe definida a função `insert :: Ord a => a -> [a] -> [a]` que dado um elemento e uma lista ordenada retorna a lista resultante de inserir ordenadamente esse elemento na lista.

35. Apresente uma definição recursiva da função `menor :: String -> String -> Bool` que dadas duas strings, retorna `True` se e só se a primeira for menor do que a segunda, segundo a ordem lexicográfica (i.e., do dicionário)

    Por exemplo, `menor "sai" "saiu"` corresponde a `True` enquanto que `menor "programacao" "funcional"` corresponde a `False`.

36. Considere que se usa o tipo `[(a,Int)]` para representar multi-conjuntos de elementos de `a`. Considere ainda que nestas listas não há pares cuja primeira componente coincida, nem cuja segunda componente seja menor ou igual a zero.

    Defina a função `elemMSet :: Eq a => a -> [(a,Int)] -> Bool` que testa se um elemento pertence a um multi-conjunto.

    Por exemplo, `elemMSet 'a' [('b',2), ('a',4), ('c',1)]` corresponde a `True` enquanto que `elemMSet 'd' [('b',2), ('a',4), ('c',1)]` corresponde a `False`.

37. Considere que se usa o tipo `[(a,Int)]` para representar multi-conjuntos de elementos de `a`. Considere ainda que nestas listas não há pares cuja primeira componente coincida, nem cuja segunda componente seja menor ou igual a zero.

    Defina a função `lengthMSet :: [(a,Int)] -> Int` que calcula o tamanho de um multi-conjunto.

    Por exemplo, `lengthMSet [('b',2), ('a',4), ('c',1)]` corresponde a `7`.

38. Considere que se usa o tipo `[(a,Int)]` para representar multi-conjuntos de elementos de `a`. Considere ainda que nestas listas não há pares cuja primeira componente coincida, nem cuja segunda componente seja menor ou igual a zero.

    Defina a função `converteMSet :: [(a,Int)] -> [a]` que converte um multi-conjuto na lista dos seus elementos

    Por exemplo, `converteMSet [('b',2), ('a',4), ('c',1)]` corresponde a `"bbaaaac"`.

39. Considere que se usa o tipo `[(a,Int)]` para representar multi-conjuntos de elementos de `a`. Considere ainda que nestas listas não há pares cuja primeira componente coincida, nem cuja segunda componente seja menor ou igual a zero.

    Defina a função `insereMSet :: Eq a => a -> [(a,Int)] -> [(a,Int)]` que acrescenta um elemento a um multi-conjunto.

    Por exemplo, `insereMSet 'c' [('b',2), ('a',4), ('c',1)]` corresponde a `[('b',2), ('a',4), ('c',2)]`.

40. Considere que se usa o tipo `[(a,Int)]` para representar multi-conjuntos de elementos de `a`. Considere ainda que nestas listas não há pares cuja primeira componente coincida, nem cuja segunda componente seja menor ou igual a zero.

    Defina a função `removeMSet :: Eq a => a -> [(a,Int)] -> [(a,Int)]` que remove um elemento a um multi-conjunto. Se o elemento não existir, deve ser retornado o multi-conjunto recebido.

    Por exemplo, `removeMSet 'c' [('b',2), ('a',4), ('c',1)]` corresponde a `[('b',2), ('a',4)]`.

41. Considere que se usa o tipo `[(a,Int)]` para representar multi-conjuntos de elementos de `a`. Considere ainda que nestas listas não há pares cuja primeira componente coincida, nem cuja segunda componente seja menor ou igual a zero.

    Defina a função `constroiMSet :: Ord a => [a] -> [(a,Int)]` dada uma lista ordenada por ordem crescente, calcula o multi-conjunto dos seus elementos.

    Por exemplo, `constroiMSet "aaabccc"` corresponde a `[('a',3), ('b',1), ('c',3)]`.

42. Apresente uma definição recursiva da função pré-definida `partitionEithers :: [Either a b] -> ([a],[b])` que divide uma lista de *Either*s em duas listas.

43. Apresente uma definição recursiva da função pré-definida `catMaybes :: [Maybe a] -> [a]` que colecciona os elementos do tipo `a` de uma lista.

44. Considere o seguinte tipo para representar movimentos de um robot.

```
data Movimento = Norte | Sul | Este | Oeste
               deriving Show
```

Defina a função `posicao :: (Int,Int) -> [Movimento] -> (Int,Int)` que, dada uma posição inicial (coordenadas) e uma lista de movimentos, calcula a posição final do robot depois de efectuar essa sequência de movimentos.

45. Considere o seguinte tipo para representar movimentos de um robot.

```
data Movimento = Norte | Sul | Este | Oeste
               deriving Show
```

Defina a função `caminho :: (Int,Int) -> (Int,Int) -> [Movimento]` que, dadas as posições inicial e final (coordenadas) do robot, produz uma lista de movimentos suficientes para que o robot passe de uma posição para a outra.

46. Considere o seguinte tipo para representar movimentos de um robot.

```
data Movimento = Norte | Sul | Este | Oeste
               deriving Show
```

Defina a função `vertical :: [Movimento] -> Bool` que, testa se uma lista de movimentos só é composta por movimentos verticais (`Norte` ou `Sul`).

47. Considere o seguinte tipo para representar a posição de um robot numa grelha.

```
data Posicao = Pos Int Int
               deriving Show
```

Defina a função `maisCentral :: [Posicao] -> Posicao` que, dada uma lista não vazia de posições, determina a que está mais perto da origem (note que as coordenadas de cada ponto são números inteiros).

48. Considere o seguinte tipo para representar a posição de um robot numa grelha.

```
data Posicao = Pos Int Int
               deriving Show
```

Defina a função `vizinhos :: Posicao -> [Posicao] -> [Posicao]` que, dada uma posição e uma lista de posições, selecciona da lista as posições adjacentes à posição dada.

49. Considere o seguinte tipo para representar a posição de um robot numa grelha.

```
data Posicao = Pos Int Int
               deriving Show
```

Defina a função `mesmaOrdenada :: [Posicao] -> Bool` que testa se todas as posições de uma dada lista têm a mesma ordenada.

50. Considere o seguinte tipo para representar o estado de um semáforo.

```
data Semaforo = Verde | Amarelo | Vermelho
                deriving Show
```

Defina a função `interseccaoOK :: [Semaforo] -> Bool` que testa se o estado dos semáforos de um cruzamento é *seguro*, i.e., não há mais do que semáforo não vermelho.