# Programação Funcional e em Lógica

## Módulo 1 – Programação Funcional

**Realizado por: Frederico Lopes (up201904580) e Pedro Pacheco (up201806824)**

## Descrição e casos teste das funções implementadas

fibRec :: (Integral a) => a -> a

Implementação recursiva pouco eficiente em que para cada termo que não 0 ou 1 é feita uma chamada recursiva para calcular esse termo. O mesmo termo pode e em princípio será calculado várias vezes.

Caso de teste:

- fibRec 50

fibLista :: (Integral a) => a -> a

Implementação com lista semelhante ao que acontece em programação dinâmica noutros tipos de linguagens. Cada termo só é calculado uma vez e este fica "guardado" na lista para ser usado no cálculo dos tempos seguintes. Só são calculados os termos necessários para o cálculo da sequência de Fibonacci.

Caso de teste:

- fibRec 500

fibListaInfinita :: (Integral a) => a -> a

Implementação com lista infinita em que se toma partido da lazy evaluation de Haskell. Uma lista infinita é gerada, que soma (zipWith) os elementos da lista com a cauda dos elementos da mesma lista criando assim uma lista com os resultados das somas.

Caso de teste:

- fibRec 1000

scanner :: String -> BigNumber

String é passada para Int e de seguida são feitas sucessivas divisões das quais é guardado o módulo para guardar no BigNumber. A função tem várias guardas para garantir que também é possível ler números negativos.

Casos de teste:

- scanner "1000"

- scanner "-1000"

output :: BigNumber -> String

Nesta função é simplesmente feita uma chamada da função "show" para todos os elementos do BigNumber através da função concatMap.

Casos de teste:

- scanner [-1,0,0,0]

- scanner [1,9,8,7]

somaBN :: BigNumber -> BigNumber -> BigNumber

A função começa por chamar a mySomaZip com os BigNumbers argumento invertidos. De seguida é chamada a função "ajustar" para corrigir os overflows. Por fim o BigNumber é invertido para a sua forma final.

Casos de teste:

- somaBN [5] [2,7]

- somaBN [5] [27]

- somaBN [9,9] [5,5]

ajustar :: BigNumber -> BigNumber

A lista com as somas dos BigNumbers invetida é passada como argumento e a função verifica se existe overflow, se existir este é passado ao algarismo mais significativo seguinte.

Casos de teste:

- ajustar [5,12,13]

- ajustar [22,7]

mySomaZip :: BigNumber -> BigNumber -> BigNumber

Soma os elementos dos dois BigNumbers sem ter em conta o overflow. Os BigNumbers podem ter diferente numero de algarismos.

Casos de teste:

- mySomaZip [9,9,9] [2,2,2]

- mySomaZip [2,3,4] [3,4,5,7]

subBN :: BigNumber -> BigNumber -> BigNumber

A função começa por chamar a mySubZip com os BigNumbers argumento invertidos. De seguida é chamada a função "ajustarSub" para corrigir os valores negativos. Por fim o BigNumber é invertido para a sua forma final.

Casos de teste:

- subBN [2,2] [9,9]

- subBN [1,1,1,1] [9,9,9]

- subBN [1,0,0] [1,0,0]

ajustarSub :: BigNumber -> BigNumber

A lista com a subtração dos BigNumbers invetida é passada como argumento e a função verifica se existe numeros menores que 0. Se existir o algarismo mais significativo seguinte é alterado.

Casos de teste:

- ajustarSub [2,-3,5]

- ajustarSub [1,1,1]

- ajustarSub [-1,-1,1]

mySubZip :: BigNumber -> BigNumber -> BigNumber

Subtrai os elementos dos dois BigNumbers sem ter em conta os numeros negativos. Os BigNumbers podem ter diferente numero de algarismos.

Casos de teste:

- mySubZip [9,8,7] [7,8,9]

- mySubZip [2,2] [9,9]

mulBN :: BigNumber -> BigNumber -> BigNumber

A função mulBN começa por colocar numa lista revertida a multiplicação simples, usando a função simpleMul, dos algarismos das duas listas representativas de BigNumbers distintos e revertidos. De seguida divide a lista anterior em sublistas e guarda as mesmas. Posteriormente adiciona um 0 no final da respetiva lista guardada consoante a sua posição na variável onde está guardada, que por sua vez é uma lista também. Os zeros são adicionados por motivos da soma que se segue posterior à multiplicação no algoritmo da multiplicação. No final, são somados os números resultantes da multiplicação, devolvendo o resultado final da multiplicação.

Casos de teste:

- mulBN [1,2,3] [4,5,6]

- mulBN [1,2] [-3,5]

- mulBN [-2,2,2] [-2,2,2,2]

somaRec :: [BigNumber] -> BigNumber

Função auxiliar chamada na função mulBN.

Executa recursivamente a soma de todos os elementos de uma lista, sendo esses elementos BigNumbers.

Casos de teste:

- somaRec [[1,2],[3,4]]

- somaRec [[1,2,3],[3,4,5]]

- somaRec [[9,9,9],[9,9,9]]

indexAt :: [BigNumber] -> Int -> [BigNumber]

Função auxiliar chamada na função mulBN.

Adiciona zeros no final de cada elemento da lista de BigNumbers (necessário para o algoritmo básico de cálculo da multiplicação) de acordo com o índice desse BigNumber na lista, logo o argumento Int que recebe será sempre 0 para começar a adicionar a partir do primeiro elemento da lista.

Casos de teste:

- indexAt [[1,2],[3,4],[5,6]] 0

addNzeros :: Int -> BigNumber -> BigNumber

Função auxiliar chamada na função indexAt que adiciona n zeros recursivamente ao final do número.

Casos de teste:

- addNzeros 3 [1,2,3]

- addNzeros 5 [1]

- addNzeros 10 [1,2,3]

splitAtLenght :: BigNumber -> Int -> [BigNumber]

Função auxiliar chamada na função mulBN. Divide um BigNumber que corresponde aos vários cálculos numéricos individuais do algoritmo básico da multiplicação e esse mesmo número em n listas (corresponde ao argumento de tipo Int), sendo esse n o número de números que irão ser somados no algoritmo básico da multiplicação.

Casos de teste:

- splitAtLenght [1,2,3,4,5,6,7,8,9] 3

- splitAtLenght [1,2,3,4,5,6] 2

- [1,2,3,4,5,6,7,8] 4

xor :: Bool -> Bool -> Bool

Função auxiliar chamada na função mulBN. Usada para ver quando um número é negativo.

Casos de teste:

- xor True False

- xor False True

- xor True True

- xor False False

changeNeg :: BigNumber -> BigNumber

Função auxiliar chamada na função mulBN. Usada para mudar um BigNumber de negativo para positivo ou vice-versa.

Casos de teste:

- changeNeg [1,2,3]

- changeNeg [-1,2,3]

checkNeg :: Int -> Int -> Bool

Função auxiliar chamada na função mulBN. Usada para verificar se dois números são ambos negativos.

Casos de teste:

- checkNeg 1 2

- checkNeg (-1) 2

- checkNeg (-1) (-2)

checkNegSolo :: Int -> Bool

Função auxiliar chamada na função mulBN. Usada para verificar se um número é negativo ou não.

Casos de teste:

- checkNegSolo 1

- checkNegSolo (-1)

simpleMul :: Int -> Int -> Int

Função auxiliar chamada na função mulBN. Usada para executar a multiplicação simples de dois números.

Casos de teste:

- simpleMul 2 3

- simpleMul 12 34

- simpleMul 543 212

divBN :: BigNumber -> BigNumber -> (BigNumber, BigNumber)

Função chama a função auxiliar divAux para calcular o quociente da divisão e utiliza esse valor para calcular o resto.

Casos de teste:

- divBN [1,0,0] [3,3]

- divBN [1,0,0] [2,5]

- divBN [9,9,9,9,9] [7]

divAux :: BigNumber -> BigNumber -> BigNumber -> BigNumber

Verifica se o dividendo é maior ou igual ao divisor, se for "incrementa" o contador (faz uma nova chamada a função com o contador incrementado) e subtrai o dividendo ao divisor. Esta operação repete-se enqaunto o dividendo for maior ou igual ao divisor

Casos de teste:

- divAux [9,9,9,9,9] [7] [0]

largerThan :: BigNumber -> BigNumber -> Bool

Verifica se o primeiro argumento é maior ou igual ao segundo

Casos de teste:

- largerThan [2,3] [3,2]

- largerThan [0] [0]

- largerThan [-1,0,0] [1,0,0]

fibRecBN :: Int -> BigNumber

Semelhante a fibRec com a diferença que a operação de soma entre termos é feita com somaBN devido aos termos serem BigNumbers.

Casos de teste:

- fibListaInfinitaBN 20

fibListaBN :: Int -> BigNumber

Semelhante a fibLista com a diferença que a operação de soma entre termos é feita com somaBN devido aos termos serem BigNumbers.

Casos de teste:

- fibListaInfinitaBN 200

fibListaInfinitaBN :: Int -> BigNumber

Semelhante a fibListaInfinita com a diferença que a operação de soma entre termos é feita com somaBN devido aos termos serem BigNumbers.

Casos de teste:

- fibListaInfinitaBN 200

safeDivBN :: BigNumber -> BigNumber -> Maybe (BigNumber, BigNumber)

Versão da função divBN com capacidade de detetar divisões por 0 em compile-time que para isso retorna *monads* do tipo Maybe.

No caso de existir uma divisão por 0 é retornado *Nothing*, em todos os outros casos é retornado *Just* (divBN) ou seja, é feita uma chamada a divBN com os mesmos argumentos de safeDivBn.

Casos de teste:

- divBN [9,9,9,9,9] [7]

- divBN [1,0,0] [0]

O desenvolvimento das funções da alínea 2 (dois) do enunciado tiveram por base uma mistura de estratégias, sendo elas recursão, condições, algoritmos de cálculo básicos e compreensões, tendo a variedade e intercalação das mesmas o objetivo de tentar implementar as funções pretendidas de uma forma variada a fim de aplicar os vários conhecimentos adquiridos.

Int -> Int : A limitação encontra-se diretamente relacionada com o limite máximo do valor de Int em Haskell (9223372036854775807). O argumento máximo que as funções

de Fibonacci aceitam sem acontecer overflow é 94, depois disso os resultados passam a ser negativos.

Integer -> Integer : A limitação encontra-se diretamente relacionada com o limite de memória da máquina onde se executa o programa.

BigNumber -> BigNumber : Não é possível existir overflow devido a implementação em listas de BigNumber e sendo assim, não terá um limite superior ou inferior de representação, sendo apenas imposta a mesma condição de Integer -> Integer, pois ao não existir mais memória disponível na máquina, não será possível continuar com a representação. Comparativamente a implementações em Integer, é menos eficiente dado Integer ser um tipo já predefinido.