**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

Nesta função é feita uma soma sucessiva dos algarismos menos significativos e se existir overflow este é passado para o algarismo seguinte. Foi necessário criar várias guardas para os casos de overflow para não ficarem escritos 0 (zeros) no BigNumber (ex: somaBN [1,5] [1,0] seria [0,2,5] sem as guardas).

Estas somas sucessivas são conseguidas a partir de chamadas recursivas para aceder aos vários elementos do BigNumber.

Casos de teste:

- somaBN [5] [2,7]

- somaBN [5] [27]

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

subBN :: BigNumber -> BigNumber -> BigNumber

A função de subtração é algo semelhante à de adição da parte de ser uma subtração sucessiva, ainda que as operações realizadas entre os algarismos sejam diferentes.

Casos de teste:

-

-

-

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]

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:

-

-

-

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:

-

-

-

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 :: 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:

-

-

-

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

Casos de teste:

-

-

-

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 relaciona com a arquitetura da máquina (32 ou 64 bits) onde se executa o programa, não podendo exceder o limite imposto pela mesma.

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.