FEUP PFL Project 2020/2021

Módulo Fib.hs

fibRec

```
fibRec :: (Integral a) => a -> a
fibRec 0 = 0
fibRec 1 = 1
fibRec x = fibRec (x-2) + fibRec (x-1)
```

A função **fibRec** retorna o elemento de índice fornecido da lista de Fibonacci através de uma implementação recursiva, chamando a própria para somar os dois elementos anteriores, parando apenas quando chegar a um dos casos base.

Testes

```
ghci> fibRec 20
6765
ghci> fibRec 10
55
ghci> fibRec 1
1
```

Como podemos observar, todos estes resultados estão de acordo com a série de Fibonacci.

fibLista

```
fibLista :: (Integral a) => a -> a
fibLista 0 = 0
fibLista 1 = 1
fibLista x = fibListaAux x 2 [0,1] !! fromIntegral x

fibListaAux :: (Integral a) => a -> a -> [a] -> [a]
fibListaAux x n acc | (x+1) == n = acc
                    | otherwise = fibListaAux x (n+1) (acc ++ [acc !!
fromIntegral (n-2) + acc !! fromIntegral (n-1)])
```

A função **fibLista** retorna o elemento de índice fornecido da sequência de Fibonacci utilizando programação dinâmica para pré-calcular uma lista de resultados parciais. Enquanto o número de elementos da lista não for igual ao índice + 1 fornecido como argumento iremos iterar a lista criando um novo elemento somando os dois anteriores.

Testes

```
ghci> fibLista 15
610
ghci> fibLista 9
34
ghci> fibLista 0
0
```

Como podemos observar, todos estes resultados estão de acordo com a sequência de Fibonacci.

fibListaInfinita

```
fibListaInfinita :: (Integral a) => a -> a
fibListaInfinita x = fib !! fromIntegral x

fib :: (Integral a) => [a]
fib = scanl (+) 0 (1 : fib)
```

A função **fibListaInfinita** retorna o elemento de índice fornecido da sequência de Fibonacci através da criação de uma lista infinita com a utilização da função **scanl** do prelude. O **scanl** irá somar o segundo argumento, o acumulador, com o primeiro elemento da lista. O resultado dessa operação irá ser colocada no acumulador, que será então somada ao segundo elemento da lista, etc... Somando a lista Fibs com ela própria concatenada com um 1 inicial produz a lista de Fibonacci.

Iteração	Acumulador	Elemento da Lista	Resultado (Acumulador + Elemento)
1	0	1	1
2	1	0	1
3	1	1	2
4	2	1	3
5	3	2	5

Testes

```
ghci> fibListaInfinita 30
832040
ghci> fibListaInfinita 60
1548008755920
ghci> fibListaInfinita 7
13
```

Como podemos observar, todos estes resultados estão de acordo com a sequência de Fibonacci.

Módulo Fib.hs com BigNumbers

fibRecBN

```
fibRecBN :: BigNumber -> BigNumber
fibRecBN [] = []
fibRecBN [1] = [1]
fibRecBN x = somaBN (fibRecBN (subBN x [1])) (fibRecBN (subBN x [2]))
```

A função **fibRecBN** segue a mesma lógica da função **fibRec** mas implementada com BigNumbers, sendo assim necessário utilizar as operações de soma e subtração entre BigNumber (**somaBN** e **subBn**, respetivamente). É utilizada uma implementação recursiva para somar os dois elementos anteriores e parar quando chegar a um dos casos base.

Testes

```
ghci> fibRecBN [2,0]
[6,7,6,5]
ghci> fibRecBN [1,0]
[5,5]
ghci> fibRecBN [1]
[1]
```

Como podemos observar, os resultados estão de acordo tanto com a sequência de Fibonacci como com a implementação da mesma função sem os BigNumbers.

fibListaBN

```
fibListaBN :: BigNumber -> BigNumber
fibListaBN x = fibListaBNAux x ([],[1])

fibListaBNAux :: BigNumber -> (BigNumber, BigNumber) -> BigNumber
fibListaBNAux [] (x,y) = x
fibListaBNAux [1] (x,y) = y
fibListaBNAux a (x,y) = fibListaBNAux (subBN a [1]) (y, somaBN x y)
```

A função **fibListaBN** utiliza uma abordagem de programação dinâmica semelhante da **fibLista**, no entanto, ao invés de armazenar uma lista de resultados parciais, guardamos apenas os dois valores anteriores. A utilização do operador !! é ímpossível sem a conversão de BigNumber para inteiro, daí optarmos por esta abordagem como alternativa.

NOTA: Seria possível associar um índice BigNumber a cada valor da lista de números Fibonacci com um tuplo (Índice, Número) e comparar posteriormente o índice, mas esta implementação seria dispendiosa em termos de memória e performance, daí optarmos por esta abordagem.

Testes

```
ghci> fibListaBN [1,5]
[6,1,0]
ghci> fibLista [9]
[3,4]
ghci> fibLista []
[]
```

Como podemos observar, os resultados estão de acordo tanto com a sequência de Fibonacci como com a implementação da mesma função sem os BigNumbers.

fibListaInfinitaBN

```
fibListaInfinitaBN :: BigNumber -> BigNumber
fibListaInfinitaBN x = head (fibListaInfinitaBNAux x)

fibListaInfinitaBNAux :: BigNumber -> [BigNumber]
fibListaInfinitaBNAux [] = fibBN
fibListaInfinitaBNAux x = tail (fibListaInfinitaBNAux (subBN x [1]))

fibBN :: [BigNumber]
fibBN = scanl somaBN [] ([1] : fibBN)
```

A função **fibListaInfinitaBN** utiliza a mesma lista infinita da função **fibListaInfinita** adaptada para BigNumbers, no entanto, dado que não é possível a utilização do operador !! utilizamos uma função auxiliar para iterar a lista infinita através da chamada recursiva da função **tail** do prelude.

Testes

```
ghci> fibListaInfinitaBN [3,0]
[8,3,2,0,4,0]
ghci> fibListaInfinitaBN [6,0]
[1,5,4,8,0,0,8,7,5,5,9,2,0]
ghci> fibListaInfinitaBN [7]
[1,3]
ghci> fibListaInfinitaBN []
[]
```

Como podemos observar, os resultados estão de acordo tanto com a sequência de Fibonacci como com a implementação da mesma função sem os BigNumbers.

Módulo BigNumber.h

A definição de BigNumber é:

```
type BigNumber = [Int]
```

Números negativos são representados em BigNumber apenas com o primeiro dígito a negativo. Exemplo: `-123` corresponde ao BigNumber `[-1,2,3]`.

O número `0` é representado em BigNumber como `[]` (lista vazia).

scanner

```
scanner :: String -> BigNumber
scanner [] = []
scanner ['0'] = []
scanner [a] = [read [a] :: Int]
scanner (a:b:cs) | a == '-'  = toggleSignalBN (removeHeaderZerosBN [read [x] ::
Int | x <- b:cs])
                 | otherwise = removeHeaderZerosBN [read [x] :: Int | x <-
a:b:cs]
```

A função **scanner** pega num número em formato de String e devolve esse mesmo número na forma de um BigNumber. Como uma String é uma lista de caracteres, basta percorrer a lista e converter os caracteres para inteiro, utilizando a função **read :: Int** do prelude. Se o primeiro elemento da lista for o sinal negativo ('-'), é efetuada a conversão da restante lista e é negado o seu primeiro elemento.

Testes

```
ghci> scanner "123"
[1,2,3]
ghci> scanner "-123"
[-1,2,3]
ghci> scanner "0"
[]
ghci> scanner "000011"
[1,1]
ghci> scanner "-00011"
[-1,1]
```

Como podemos observar, a função comporta-se como seria esperado.

output

```
output :: BigNumber -> String
output [] = ['0']
output (x:xs) | x < 0 = '-' : bignumber
              | otherwise = bignumber
              where bignumber = foldr (\a b -> show (abs a) ++ b) [] (x:xs)
```

A função **output** faz o inverso da função **scanner**, ou seja, recebe um BigNumber e retorna o mesmo em formato de String. Dado que uma String é uma lista de caracteres, basta fazer conversão do tipo Int para Char, utilizando a função **show** do prelude, percorrendo a lista com **foldr** também disponível no **prelude**. Se o primeiro elemento do BigNumber for negativo, é só adicionar um '-' à String resultado.

Testes

```
ghci> output [1,2,3]
"123"
ghci> output [-1,2,3]
"-123"
ghci> output []
"0"
```

Como podemos observar, a função retorna os resultados esperados.

---

somaBN

```
somaBN :: BigNumber -> BigNumber -> BigNumber
somaBN [] [] = []
somaBN xs [] = xs
somaBN [] ys = ys
```

```
somaBN (x:xs) (y:ys) | x < 0 && y > 0 = subBN (y:ys) (toggleSignalBN (x:xs))
                     | x > 0 && y < 0 = subBN (x:xs) (toggleSignalBN (y:ys))
                     | x > 0 && y > 0 = somaBNAux (reverse (x:xs)) (reverse
(y:ys)) 0
                     | otherwise = toggleSignalBN (somaBNAux (reverse
(toggleSignalBN (x:xs))) (reverse (toggleSignalBN (y:ys))) 0)


somaBNAux :: BigNumber -> BigNumber -> Int -> BigNumber
somaBNAux [] [] 0             = []
somaBNAux [] [] carry         = [carry]
somaBNAux [] bs 0             = reverse bs
somaBNAux [] (b:bs) carry     = somaBNAux [] bs ((b + carry) `div` 10) ++ [(b +
carry) `mod` 10]
somaBNAux as [] 0             = reverse as
somaBNAux (a:as) [] carry     = somaBNAux as [] ((a + carry) `div` 10) ++ [(a +
carry) `mod` 10]
somaBNAux (a:as) (b:bs) carry = somaBNAux as bs ((a + b + carry) `div` 10) ++
[(a + b + carry) `mod` 10]
```

A função **somaBN** implementa a soma entre dois BigNumbers, retornando o BigNumber resultante.

Para a soma com dois BigNumbers positivos, foi utilizada uma função auxiliar **somaBNAux** que recebe como argumentos os dois BigNumbers revertidos (para facilitar a implementação) e também um inteiro que representa o carry (este ou será 0 ou 1, daí não ser ele próprio um BigNumber). Assim, soma-se elemento com elemento os dois BigNumbers, a divisão por 10 do resultado dessa soma é guardada para o retorno e chama-se recursivamente a mesma função com o restante dos dois BigNumbers com o carry calculado através do resto da divisão do resultado da soma por 10.

Para a soma com dois BigNumbers negativos é efetuada a seguinte estratégia: negando os dois BigNumbers para obtermos uma soma entre dois positivos e depois negar o resultado final. É utilizada a função auxiliar **toggleSignalBN**, que altera o sinal de um BigNumber de positivo para negativo ou vice-versa.

Para a soma de um BigNumber positivo com um negativo, é utilizada a função que implementa a subtração de dois BigNumbers, a **subBN**, cujo funcionamento será explicado no seu tópico. Assim, transforma-se a soma de um positivo com um negativo numa subtração entre dois positivos.

Testes

```
ghci> somaBN [1,2,3] [4,9]
[1,7,2]
ghci> somaBN [1,2,3] [-4,9]
[7,4]
ghci> somaBN [-1,2,3] [-4,9]
[-1,7,2]
ghci> somaBN [-1,2,3] [4,9]
[-7,4]
```

Como podemos observar, os resultados obtidos são os esperados.

subBN

```
subBN :: BigNumber -> BigNumber -> BigNumber
subBN [] [] = []
subBN xs [] = xs
subBN [] ys = toggleSignalBN ys
subBN (x:xs) (y:ys) | (x:xs) == (y:ys) = []
                    | x < 0 && y > 0 = toggleSignalBN (somaBN (toggleSignalBN
(x:xs)) (y:ys))
                    | x > 0 && y < 0 = somaBN (x:xs) (toggleSignalBN (y:ys))
                    | x < 0 && y < 0 = subBN (toggleSignalBN (y:ys))
(toggleSignalBN (x:xs))
                    | greaterThanBN (x:xs) (y:ys) =
removeHeaderZerosBN(subBNAux (reverse bn1) (reverse bn2))
                    | otherwise = toggleSignalBN (removeHeaderZerosBN (subBNAux
(reverse bn2) (reverse bn1)))
                    where bn1 = addHeaderZerosBN (length (y:ys) - length
(x:xs)) (x:xs)
                          bn2 = addHeaderZerosBN (length (x:xs) - length
(y:ys)) (y:ys)


subBNAux :: BigNumber -> BigNumber -> BigNumber
subBNAux [] [] = []
subBNAux xs [] = reverse xs
subBNAux [] ys = toggleSignalBN (reverse ys)
subBNAux (x:xs) (y:ys) | x < y = subBNAux xs ((head ys + 1) : tail ys) ++ [x +
10 - y]
                       | otherwise = subBNAux xs ys ++ [x - y]
```

A função **subBN** implementa a subtração entre dois BigNumbers, retornando o BigNumber resultante.

A função **subBN** em si faz o tratamento de sinais apenas, deixando o cálculo da subtração para a sua função auxiliar, **subBNAux**. Aplicamos na função subBN a seguinte estratégia:

- Caso apenas o primeiro número seja negativo iremos chamar a função **somaBN** com o seu absoluto e trocamos o sinal final;
- Caso apenas o segundo número seja negativo iremos chamar a função **somaBN** com o seu absoluto;
- Caso os dois números sejam negativos iremos chamar a função recursivamente com o absoluto dos dois números e pela ordem inversa dos argumentos;
- Caso os dois números sejam positivos iremos verificar, primeiramente, qual o maior em valor absoluto, de seguida subtraímos o maior ao menor utilizando a função **subBNAux** (argumentos recebidos pela ordem inversa), e posteriormente colocamos o sinal do de maior valor absoluto.

A função **subBNAux** subtrai dígito a dígito, do menos significativo para o mais significativo, verificando se o dígito do primeiro argumento é superior ao dígito do segundo argumento. Caso não seja, antes da subtração somamos 10 ao dígito do primeiro argumento e somamos 1 ao dígito seguinte do segundo argumento.

Testes

```
ghci> subBN [1,2,3] [4,9]
[7,4]
ghci> subBN [1,2,3] [-4,9]
[1,7,2]
ghci> subBN [-1,2,3] [4,9]
[-1,7,2]
ghci> subBN [-1,2,3] [-4,9]
[-7,4]
```

Como podemos observar, os resultados são os esperados.

mulBN

```
mulBN :: BigNumber -> BigNumber -> BigNumber
mulBN [] [] = []
mulBN xs [] = []
mulBN [] ys = []
mulBN (x:xs) (y:ys) | x < 0 && y > 0 = toggleSignalBN (reverse (mulBNAux
(reverse (toggleSignalBN (x:xs))) (reverse (y:ys))))
                    | x > 0 && y < 0 = toggleSignalBN (reverse (mulBNAux
(reverse (x:xs)) (reverse (toggleSignalBN (y:ys)))))
                    | x < 0 && y < 0 = reverse (mulBNAux (reverse
(toggleSignalBN (x:xs))) (reverse (toggleSignalBN (y:ys))))
                    | otherwise = reverse (mulBNAux (reverse (x:xs)) (reverse
(y:ys)))


mulBNAux :: BigNumber -> BigNumber -> BigNumber
mulBNAux [] [] = []
mulBNAux xs [] = []
mulBNAux [] ys = []
mulBNAux (x:xs) (y:ys) = reverse (somaBNAux (mulLineBN y (x:xs) 0)
(addHeaderZerosBN 1 (mulBNAux (x:xs) ys)) 0)


mulLineBN :: Int -> BigNumber -> Int -> BigNumber
mulLineBN _ [] 0 = []
mulLineBN _ [] carry = [carry]
mulLineBN x (y:ys) carry = ((x * y + carry) `mod` 10) : mulLineBN x ys ((x * y
+ carry) `div` 10)
```

A função **mulBN** efetua a multiplicação entre dois BigNumbers e retorna o BigNumber resultante.

A **mulBN** apenas é responsável por organizar os BigNumbers com os quais vai ser realizada a multiplicação, trocar os sinais (se necessário), e reverter os BigNumbers de modo a facilitar os cálculos. A multiplicação em si é realizada pela função auxiliar **mulBNAux**, que por sua vez é auxiliada pela **mulLineBN**.

A função **mulLineBn** recebe um Int que corresponde ao fator pelo qual estamos a multiplicar o BigNumber, sendo este o segundo argumento. Assim, é realizada a multiplicação entre dígitos e lidamos com o carry de forma semelhante à soma. A função é chamada recursivamente até percorrer todos os dígitos do BigNumber.

A função **mulBNAux** é responsável por chamar a **mulLineBN** para cada dígito do segundo fator, e, posteriormente, somar as linhas depois de adicionado um shift à direita de um dígito a cada linha com o recurso da função **addHeaderZerosBN** (que coloca um número arbitrário de 0's à cabeça do seu argumento).

Testes

```
ghci> mulBN [1,2,3] [4,9]
[6,0,2,7]
ghci> mulBN [1,2,3] [-4,9]
[-6,0,2,7]
ghci> mulBN [-1,2,3] [4,9]
[-6,0,2,7]
ghci> mulBN [-1,2,3] [-4,9]
[6,0,2,7]
ghci> mulBN [-1,2,3] []
[]
ghci> mulBN [5] [5]
[2,5]
```

Como podemos observar, os resultados são os esperados.

divBN

```
divBN :: BigNumber -> BigNumber -> (BigNumber, BigNumber)
divBN [] ys = ([], [])
divBN (x:xs) ys = let r = divBNAux xs ys [] [x] in (removeHeaderZerosBN (fst
r), snd r)


divBNAux :: BigNumber -> BigNumber -> BigNumber -> BigNumber -> (BigNumber,
BigNumber)
divBNAux [] ys q r = let a = slowDivBN r ys in (q ++ fst a, snd a)
divBNAux (x:xs) ys q r = let a = slowDivBN r ys in divBNAux xs ys (q ++ fst a)
(snd a ++ [x])


slowDivBN :: BigNumber -> BigNumber -> (BigNumber, BigNumber)
slowDivBN xs ys = (if null (fst r) then [0] else fst r, subBN xs (snd r))
            where r = head (filter (\x -> greaterThanBN (somaBN (snd x) ys) xs)
(timeTableBN ys))
```

A função **divBN** efetua a divisão entre dois BigNumbers positivos, retornando o resultado na forma (quociente, resto), também estes representados através de BigNumbers.

A função **divBN** vai utilizar a função **divBNAux**, que, por sua vez, vai utilizar a função **slowDivBN**.

A função **slowDivBN** determina a divisão entre dois BigNumbers positivos, no entanto, como o nome indica, é lenta, já que esta procura quantas vezes o segundo argumento consegue pertencer ao primeiro argumento (utilizando a função **timeTableBN**). Para optimizar os cálculos fazemos uma divisão dígito a dígito na função **divBNAux**.

A função **divBNAux** vai realizar vários cálculos intermédios ao longo do seu primeiro argumento, o dividendo, dígito a dígito. Procura dividir o primeiro dígito do dividendo pelo divisor (segundo argumento) e guarda o resto desta divisão num dos seus dois acumuladores, neste caso o r, concatenado com o próximo dígito do dividendo, e guarda o quociente da divisão anterior (`[]` sendo a primeira iteração) concatenado com o quociente desta divisão no outro acumulador, o q. Posteriormente chamamos esta função recursivamente com a cauda do dividendo até que o dividendo seja esgotado.

Dado que a divisão pode resultar num quociente `0` e a nossa representação de `0` ser `[]`, de modo a que o quociente tenha `0's` a meio tivemos que, provisoriamente, representar o `0` por `[0]` na função slowDivBN, o que traduz-se, também, na utilização da função **removeHeaderZerosBN** em **divBN** de modo a não representar `0's` no início do resultado.

Testes

```
ghci> divBN [1,2,3] [4,9]
([2],[2,5])
ghci> divBN [1,2,3] [1]
([1,2,3],[])
ghci> divBN [4,9] [1,2,3]
([],[4,9])
```

Como podemos observar, os resultados são os esperados.

safeDivBN

```
safeDivBN :: BigNumber -> BigNumber -> Maybe (BigNumber, BigNumber)
safeDivBN xs [] = Nothing
safeDivBN xs ys = Just (divBN xs ys)
```

A função **safeDivBN** realiza também a divisão, mas esta é capaz de detetar a divisão por zero em compile-time. Assim, são retornados monads do tipo Maybe: Nothing quando é efetuada uma divisão por zero e Just com o resultado da divisão caso não seja.

Testes

```
ghci> safeDivBN [1,2,3] []
Nothing
ghci> safeDivBN [1,2,3] [4,9]
Just ([2],[2,5])
```

Como podemos observar, os resultados são os esperados.

Funções Auxiliares ao Módulo BigNumber.hs

addHeaderZerosBN

```
addHeaderZerosBN :: Int -> BigNumber -> BigNumber
addHeaderZerosBN x bn = [0 | _ <- [1..x]] ++ bn
```

A função **addHeaderZerosBN** adiciona a quantidade indicada de 0's à cabeça de um BigNumber.

removeHeaderZerosBN

```
removeHeaderZerosBN :: BigNumber -> BigNumber
removeHeaderZerosBN [] = []
removeHeaderZerosBN (x:xs) | x /= 0 = x:xs
                           | otherwise = removeHeaderZerosBN xs
```

A função **removeHeaderZerosBN** remove os 0's iniciais de um BigNumber.

toggleSignalBN

```
toggleSignalBN :: BigNumber -> BigNumber
toggleSignalBN [] = []
toggleSignalBN (x:xs) | x == 0 = x : toggleSignalBN xs
                      | otherwise = (-x):xs
```

A função **toggleSignalBN** troca o sinal a um BigNumber.

greaterThanBN

```
greaterThanBN :: BigNumber -> BigNumber -> Bool
greaterThanBN [] [] = False
greaterThanBN (x:xs) [] | x < 0 = False
                        | otherwise = True
greaterThanBN [] (y:ys) | y < 0 = True
                        | otherwise = False
greaterThanBN (x:xs) (y:ys) | x < 0 && y < 0 = toggleSignalBN bn1 <
toggleSignalBN bn2
                            | otherwise = bn1 > bn2
                            where bn1 = addHeaderZerosBN (length (y:ys) -
length (x:xs)) (x:xs)
                                  bn2 = addHeaderZerosBN (length (x:xs) -
length (y:ys)) (y:ys)
```

A função **greaterThanBN** verifica se o BigNumber passado como primeiro argumento é maior do que o BigNumber passado como segundo argumento.

timeTableBN

```
timeTableBN :: BigNumber -> [(BigNumber, BigNumber)]
timeTableBN x = scanl (\a b -> (somaBN (fst a) [1], somaBN (snd a) b)) ([], [])
[x | _ <- [1..]]
```

A função **timeTableBN** retorna a tabuada do BigNumber passado como argumento, no formato [(Indice, Multiplicação)].

Por exemplo:

```
ghci> take 5 (timeTableBN [2])
[([],[]),([1],[2]),([2],[4]),([3],[6]),([4],[8])]
```

Alínea 4

Tipo	Mínimo	Máximo
Int *	-2^29	2^29-1
Integer	- Inf	+ Inf
BigNumber	- Inf	+ Inf

* De acordo com a documentação de Haskell o tipo Int tem limite inferior e superior indicados na tabela, no entanto, como podemos ver pelo bloco de código abaixo, dado que o código foi compilado numa arquitetura 64 bits, os nossos limites são, realmente, `[-2^63, -2^63-1]`.

```
ghci> (minBound, maxBound) :: (Int, Int)
(-9223372036854775808,9223372036854775807)
```

Deste modo, a implementação de todas as funções que trabalham com Int são viáveis apenas até a um argumento igual ou inferior a 92, pois o valor do número de Fibonacci de índice 93 ultrapassa o limite representável por esse tipo, como podemos ver abaixo.

```
ghci> head (filter (\x -> snd x < 0) [(x, fibLista x::Int) | x <- [0..]])
(93,-6246583658587674878)
```

A implementação das funções **fibLista** e **fibListaInfinita** aplicadas a Integers estão restringidas pelo operador !! que recebe um número do tipo Int como índice. Deste modo o argumento destas funções tem um limite superior de 2^63-1, após esse valor ocorrerá overflow de índices, inviabilizando os resultados. A função **fibRec** aplicada a Integer não é afetada, pois esta não utiliza o operador !!, e, deste modo, o seu argumento pode, hipoteticamente, ser infinito, sendo este apenas restringido pela memória da própria máquina.

Na implementação das funções aplicadas a BigNumbers dado que não utilizam o operador !! e este tipo não ter limites, em teoria, conseguem representar números de Fibonacci infinitos, enquanto a memória da própria máquina suportar.

Grupo G2_07

- Adriano Soares up201904873@up.pt
- Vasco Alves up201808031@up.pt