feup-pfl-2021

Programação Funcional e Lógica

G6_04

- André Pereira - up201905650
- Matilde Oliveira - up201906954

IMPORTANT

In BigNumber representation, if the digit 0 appears at the beginning of the array, the BigNumber is negative.

To represent the number 0 in BigNumber notation we use the empty array '[]'.

In string representation the negative numbers must have the '-' char at the beginning.

Functions

Function	fibRec
Usage Cases	We tested for integer small and larger numbers and it does not work for negative numbers. Note that it becames harder to calculate in large numbers.
Function Description	The function adopts a **recursive strategy** because every time it needs sum the previous two numbers of the sequence it calls the same function recursively with those two parameters. We just defined the first two elements of the sequence (0 and 1) and then it calculates the *nth* one.

Function	fibLista
Usage Cases	We tested for big and larger integer numbers and it does not work for negative numbers. Some conclusions about the results above 92 are presented in exercise 4. It is the fastest between the three aproaches but with larger numbers it also costs time.
Function Description	The function adopts a **dynamic programming** aprouch because he starts to calculate the finobaci numbers from 0 until the *nth* number asked, dynamically saving the previous two calculated finobacci numbers to produce the next one.

Function	fibListaInfinita
Usage Cases	We tested for integer small and larger numbers and it does not work for negative numbers. Note that it becames harder to calculate in large numbers.
Function Description	The function adopts a **infinite list** strategy since he builds a infinite list with all the finobaci numbers. Thanks to haskell properties by asking the *nth* number he self stops the infite list calculation on the asked index

Function	fibRecBN
Usage Cases	We tested for positive `BigNumber` numbers not too high because it starts to be hard to calculate and it does not work for negative numbers.
Function Description	Same as the integral version. The function adopts a **recursive strategy** because every time it needs sum the previous two numbers of the sequence it calls the same function recursively with those two parameters. We just defined the first two elements of the sequence (0 and 1) and then it calculates the *nth* one.

Function	fibListaBN
Usage Cases	We tested for each `BigNumber` number till x and it does not work for negative numbers.
Function Description	Same as the integral version. The function adopts a **dynamic programming** aprouch because he starts to calculate the finobaci numbers from 0 until the *nth* number asked, dynamically saving the previous two calculated finobacci numbers to produce the next one.

Function	fibListaInfinitaBN
Usage Cases	We tested for each `BigNumber` number till x and it does not work for negative numbers.
Function Description	Same as the integral version. The function adopts a **infinite list** strategy since he builds a infinite list with all the finobaci numbers. Thanks to haskell properties by asking the *nth* number he self stops the infite list calculation on the asked index

Function	scanner
Usage Cases	We tested scanner with positive, negative and the number zero.
Function Description	This function was made to facilitate the conversion from `String` to `BigNumber`.
Implementation	This function just goes through every element of the `String`, `Char`'s, and converts them into Int putting them in a list that represents the `BigNumber`. This aproach is done with a recursive strategy in the function `scannerHelp` leaving the function `scanner` just calling them if there is no error in the input. The conversion from `Char` to `Int` is done with the function `digitToInt` from the `Data.Char` module.

Function	output
Usage Cases	We tested scanner with positive, negative and the number zero.

Function	output
Function Description	This function was made to facilitate the conversion from `BigNumber` to `String`.
Implementation	This function just goes through every element of the `BigNumber` (list of `Int`) and converts them into `Char` putting them in a list that represents the final number in a `String`. The conversion from `Int` to `Char` is done with the function `intToDigit` from the `Data.Char` module.

Function	somaBN
Usage Cases	We tested this function with all kinds of combinations like, positive + positive `somaBN [1,0] [2,0] => [3,0]`, positive + negative `somaBN [1,0] [0,2,0] => [0,1,0]`, sum of numbers that result in more decimal houses `somaBN [8,8] [1,3] => [1,0,1]`, sum of numbers that result in the reduction of decimal houses `somaBN [1,5] [0,1,3] => [2]`, sum with 0 `somaBN [1,5] [] => [1,5]` and sum resulting in 0 `somaBN [0,1,3] [1,3] => []`
Function Description	This function takes two `BigNumbers` returning their sum
Implementation	We start by inverting the `BigNumber` received to start the adiction operation with the units digit in function `somaBN`. Next, with the help of the `soma` we add the first element of each `BigNumber array`. If the sum is bigger than 10, by making the `mod` of 10, we make sure that we only have single digit numbers in our array. Then we make the `div` of 10 and add it to the next element of the `BigNumber`. All the patterns present are intended to cover all case scenarios if one of the `BigNumbers` is bigger than the other, this in function `somaBN`. In order to deal with the negative numbers we analyse the numbers given at the start and if the signals are opposites we treat the request as an subtration of `BigNumbers` with function `subBN`.

Function	subBN
Usage Cases	We tested this function with all kinds of combinations like, positive - positive `subBN [1,0] [2,0] => [0,1,0]`, positive - negative `subBN [1,0] [0,2,0] => [3,0]`, negative - positive `subBN [0,1,0,0] [2,0] => [0,1,2,0]`, negative - negative `subBN [0,1,0,0] [0,2,0] => [0,8,0]`, subtration with 0 `subBN [1,5] [] => [1,5]` and subtration resulting in 0 `subBN [1,3] [1,3] => []`
Function Description	This function takes two BigNumbers returning their subtration

Function	subBN
Implementation	We start by inverting the BigNumber received to start the operation before calling function `sub` that helps with the procedure described to effectuate the subtraction. Next we verify wich of the given numbers is bigger, by making this step we reduce the number of patterns necessary and we can safely decide wich is the result signal. After these verification steps, in auxiliary function `sub`, we start the operation, just like the sum, we make use of the `div` and `mod` operations to ensure that he only have single house digits in our number representation and the subtration is excess is correctly reduced to the next digits. In this opperation when the signals, of the given numbers are different, we make use the sum function `somaBN`.

Function	mulBN
Usage Cases	We tested this function with all interesting combinations for multiplication, positive * positive `mulBN [4,3] [2,3] => [9,8,9]`, positive * negative `mulBN [0,1,2,3] [1,2,3] => [0,1,5,1,2,9]`, negative * negative `mulBN [0,4,3] [0,1,2,1] => [5,2,0,3]`, also multiplication by 0 is covered in `mulBN [2,3,4,2] [] => []`
Function Description	This function takes two BigNumbers returning their multiplication
Implementation	To implement this function we first start by inverting the two given BigNumbers. In order to make the multiplication operation, in function `mul`, we take an recursive aprouch since we sum, the result of the multiplication of all the digits of the first `BigNumber` with the first digit of the second `BigNumber`, with the value returned from the recursive call of the `mulBN` with the first `BigNumber` and the tail of the second `BigNumber`. To deal with the negative numbers we verify if the numbers have both the same signal, if not then we add the zero at the beginning that represents negative numbers.

Function	divBN
Usage Cases	We tested this function with all the possibel combinations. Positive divided by positive `divBN [1,2,3] [2,3] => ([5],[8])`, positive divided by negative `divBN [5,3,3] [0,5,3] => ([0,1,1],[0,5,0])`, negative divided by positive `divBN [0,7,3] [7] => ([0,1,1],[4])`, negative divided by negative `divBN [0,2,4,3] [0,8] => ([3,0],[0,3])`. We also tested values where the remainder is zero `divBN [2,5,6] [8] => ([3,2],[])`, when the dividend is smaller then the divisor `divBN [2,3] [5,3] => ([],[2,3])`, when the dividend is zero `divBN [] [4] => ([],[])` and lastely we tested the case when the divisor is `zero divBN [5] [] => "Exception: divide by zero"`
Function Description	This function takes two `BigNumbers` returning a tuple with quotient and the remainder. The division with negative numbers is made just like the `mod` and `div` operators from haskell.

Function	divBN
Implementation	In order to implement this function, we implemented function `divi2` that, removes from the beginning of the dividend, a number and append it to the final of the current auxiliar `BigNumber`. If the auxiliar `BigNumber` is bigger then the divisor we calculate the times that the divisor must by multiplied to reach the auxiliar `BigNumber`(the function `divi` assist this calculation). The result is appendeded to the resulting `BigNumber` and the remainer is mantained in the auxiliar `BigNumber`. We repeat the process until we reach the end of the dividend. If we add an 0 to the auxiliar `BigNumber` and its current value is [] we must add a 0 to the result. The `divBN` function deals with negative numbers and the correspondent results of the known `mod` and `div` functions we had the need to make extra operations to the final output of the division to make the remainer negative or not and calls the previous descripted function `divi2`.

Function	safeDivBN
Usage Cases	We test this function with all the cases described in `divBN safeDivBN [1,6] [3] => Just ([5],[1])`. We also test it when the divisor is zero `safeDivBN [1,6] [] => Nothing`
Function Description	This function takes two `BigNumbers` returning their division safely
Implementation	The implementation of this function just calls the `divBN` if the second argument is not zero, just returning Nothing if not.

Ex4

To come up with a conclusion for this exercise we searched for Maximum representation of either `Int` or `Integer`.

We noticed that `Int` type has a maximum representation even if `Integer` does not.

For the representation with 64-bits like ghci the maximum Int representation is: `2^63-1` (that corresponds to -`9223372036854775807`).

Hence, the maximum result of fibLista that we can have being an `Int` is for the argument: `92`. `fibLista 92 => 7540113804746346429`

For the `93` argument if we don´t specify we want it to be an `Int` it is automatically converting and giving the result in `Integer`. `fibLista 93 :: Int => -6246583658587674878 (overflow)` `fibLista 93 => 12200160415121876738 (integer)(correct)`

Int Representation:

- `12200160415121876738 - fib 93`
- `9223372036854775807 - max int` - `7540113804746346429 - fib 92`

On the other side, Integer representation has prevention from overflow, this ables us to represent every number until de computer memory does not allow any more.

For the `BigNumber` functions as the numbers are represented in a list of `Int`, the representation is also infinit. Each element of the list, an `Int`, never is more than 9 so it does not have any limitation. As for the size of the list, it may be infinit as haskell allows this representation, but can not go after the memory size.

Even if the two last representations: `Integer` and `BigNumbers` can represent every number. This function calls can be tiring and long lasting, even using dynamic programming stratagies. The BigNumber is even more than the Integer one because of the extra operations it needs to do, such as reverse.