PFL TP1 2021 2022

Trabalho realizado por:

Diogo André Barbosa Nunes	up201808546
Margarida Alves Pinho	up201704599

Descrição de casos de teste para todas as funções

Na nossa implementação, um BigNumber é composto por: - *Positive* (se for positivo) ou *Negative* (se for negativo); - Lista de algarismos do número em questão.

```
data BigNumber = Positive [Int] | Negative [Int] deriving (Show)
```

Fib.hs

função	arg	resultado
fibRec	10	55
fibLista	20	6765
fibListaInfinita	30	832040
fibRecBN	(Positive [2,0])	Positive [6,7,6,5]
fibListaBN	(Positive [3,0])	Positive [8,3,2,0,4,0]
fibListaInfinitaBN	(Positive [1,0])	Positive [5,5]

função	resultado
fibs	[0,1,1,2,3,5,8,13,21,34,55,...]
fibsBN	[Positive [0],Positive [1],Positive [1],Positive [2],Positive [3],Positive [5],Positive [8],Positive [1,3],Positive [2,1],Positive [3,4],Positive [5,5],...]

BigNumber.hs

Funções pedidas:

função	arg	resultado
scanner	"123"	Positive [1,2,3]
scanner	"-123"	Negative [1,2,3]
scanner	"0"	Positive [0]
output	(Positive [1,2,3])	"123"
output	(Negative [1,2,3])	"-123"

função	arg1	arg2	resultado
somaBN	(Positive [1,2,3])	(Positive [4,5,6])	Positive [5,7,9]
somaBN	(Positive [1,2,3])	(Negative [4,5,6])	Negative [3,3,3]

função	arg1	arg2	resultado
somaBN	(Negative [1,2,3])	(Positive [4,5,6])	Positive [3,3,3]
somaBN	(Negative [1,2,3])	(Negative [4,5,6])	Negative [5,7,9]
--	--	--	--
subBN	(Positive [1,2,3])	(Positive [4,5,6])	Negative [3,3,3]
subBN	(Positive [1,2,3])	(Negative [4,5,6])	Positive [5,7,9]
subBN	(Negative [1,2,3])	(Positive [4,5,6])	Negative [5,7,9]
subBN	(Negative [1,2,3])	(Negative [4,5,6])	Positive [3,3,3]
--	--	--	--
mulBN	(Positive [1,2,3])	(Positive [4,5,6])	Positive [1,5,1,2,9]
mulBN	(Positive [1,2,3])	(Negative [4,5,6])	Negative [1,5,1,2,9]
--	--	--	--
divBN	(Positive [1,2,0])	(Positive [1,0,0])	(Positive [1],Positive [2,0])
divBN	(Positive [1,2,0])	(Negative [1,0,0])	(Negative [1],Negative [2,0])
divBN	(Negative [1,2,0])	(Negative [1,0,0])	(Positive [1],Positive [2,0])
divBN	(Positive [1,2,0])	(Positive [0])	Exception: Infinity
--	--	--	--
safeDivBN	(Positive [1,2,0])	(Positive [1,0,0])	Just (Positive [1],Positive [2,0])
safeDivBN	(Positive [1,2,0])	(Positive [0])	Nothing

Funções auxiliares de 1 argumento:

função	arg	resultado
removerZerosEsquerdaBN	(Positive [0,0,0,1,2,3])	Positive [1,2,3]
mudarSinalBN	(Positive [1,2,3])	Negative [1,2,3]
mudarSinalDivBN	(Positive [1,2,3],Positive [4,5,6])	(Negative [1,2,3],Negative [4,5,6])
reverseBN	(Positive [1,2,3])	Positive [3,2,1]
carryBN	(Positive [1,12,1])	Positive [2,2,1]
carryBN	(Positive [9,87,6])	Positive [1,7,7,6]

Funções auxiliares de 2 argumentos:

função	arg1	arg2	resultado

função	arg1	arg2	resultado
maiorBN	(Positive [1,2,3])	(Positive [4,5,6])	False
maiorBN	(Positive [1,2,3])	(Negative [4,5,6])	True
maiorBN	(Negative [1,2,3])	(Positive [4,5,6])	False
maiorBN	(Positive [1,2,3])	(Positive [4,5])	True
maiorBN	(Positive [1,2,3])	(Positive [0,0,0,0,4,5])	True
maiorBN	(Positive [1,2,3])	(Positive [1,2,3])	False
removeBN	"-123"	'-'	"123"
appendBN	1	(Positive [2,3,4])	Positive [1,2,3,4]
adicionarZerosBN	(Positive [1,2,3])	2	Positive [1,2,3,0,0]

Explicação sucinta do funcionamento de cada função

Fib.hs

Funções pedidas

- **fibRec**: Função recursiva que tem um número "n" como argumento e calcula o n-ésimo número de Fibonacci.
- **fibLista**: Função que tem um número "n" como argumento e retorna o n-ésimo número de Fibonacci usando uma lista de resultados parciais (programação dinâmica).
- **fibListaInfinita**: Função que tem um número "n" como argumento e retorna o n-ésimo número de Fibonacci numa lista infinita com todos os números de Fibonacci (fibs).
- **fibRecBN**: Função recursiva que tem um BigNumber "n" como argumento e calcula o n-ésimo número de Fibonacci.
- **fibListaBN**: Função que tem um BigNumber "n" como argumento e calcula o n-ésimo número de Fibonacci usando uma lista de resultados parciais (progração dinâmica).
- **fibListaInfinitaBN**: Função que tem um BigNumber "n" como argumento e retorna o n-ésimo número de Fibonacci numa lista infinita com todos os números (BigNumber) de Fibonacci (fibsBN).

Funções auxiliares

- **fibs**: Função que cria uma lista infinita com os números de Fibonacci.
- **fibsBN**: Função que cria uma lista infinita com os números (BigNumber) de Fibonacci.
- **indexAtBN**: Função que retorna o n-ésimo número (BigNumber) numa lista (de BigNumber's). Equivalente ao uso de (!!) com inteiros.

BigNumber.hs

Funções pedidas

- **scanner**: função que recebe uma string e retorna um BigNumber;
- **output**: função que recebe um BigNumber e retorna uma string;
- **somaBN**: função que recebe 2 BigNumber's e retorna a sua soma;
- **subBN**: função que recebe 2 BigNumber's e retorna a sua diferença;
- **mulBN**: função que recebe 2 BigNumber's e retorna a sua multiplicação;
- **divBN**: função que recebe 2 BigNumber's e retorna um tuplo constituído por quociente e resto;
- **safeDivBN**: função equivalente a divBN mas capaz de detetar divisões por zero (retornando Nothing).

Funções auxiliares

- **removerZerosEsquerdaBN**: remoção dos zeros iniciais ("à esquerda") de um BigNumber.
- **mudarSinalBN**: mudança do prefixo de sinal de um BigNumber.
- **mudarSinalDivBN**: mudança de sinal dos 2 elementos de um par de BigNumber's (usado para divBN onde um dos elementos é negativo).
- **auxMaiorBN**: função que, dados 2 BN positivos, verifica qual deles é maior, tendo em conta o tamanho da lista e só depois o valor de cada algarismo.
- **maiorBN**: função que retorna True se um BigNumber for maior que outro.
- **removeBN**: remoção de um elemento/caractere de uma lista/string (usado no scanner, para remover o sinal "menos" numa string).
- **appendBN**: função que dá *append* de um elemento a uma lista (função a:b mas adaptada a BigNumber's (para ter em conta o prefixo de sinal).
- **reverseBN**: função que dá *reverse* à lista de algarismos de um BigNumber e preserva o sinal.
- **auxSomaBN**: função que calcula verdadeiramente o valor da soma, a partir dos argumentos invertidos (algarismos das unidades à cabeça das listas de algarismos).
- **auxSubBN**: função que calcula verdadeiramente o valor da diferença, a partir dos argumentos invertidos.
- **adicionarZerosBN**: função que adiciona zeros à direita para alterar a ordem de grandeza.
- **auxMulBN**: função que calcula verdadeiramente o valor da multiplicação, a partir dos argumentos invertidos.
- **auxCarryBN**: função que auxilia o cálculo em carryBN, tendo em conta o valor do carry a ser transportado para o conjunto de algarismos de ordem de

grandeza seguinte.
- **carryBN**: função que separa uma lista de números numa lista de algarismos de um BigNumber (1 único algarismo por elemento da lista).
- **carryPairBN**: função com o mesmo objetivo de carryBN, mas aplicada a um par de BigNumber's.
- **auxDivBN**: função que verifica se o dividendo é maior ou igual ao divisor. Se sim, chama recursivamente a mesma função com o dividendo e o quociente "atualizados". Caso contrário, chama carryBN para fazer a separação em algarismos.

Estratégias utilizadas na implementação das funções da alínea 2

Seguindo a ordem do enunciado, a nossa função **scanner** recebe uma string e a primeira verificação que faz é confirmar que o primeiro caractere dessa string é ou não o sinal "-". Se for, vai ser construído um BigNumber negativo com a lista de algarismos que vão sendo lidos (usando a função map), mas a string a ser usada no map precisa da remoção do caractere "-". Caso não tenha esse caractere, não precisa desta remoção e apenas cria um BigNumber positivo usando a função map com a string inicial.

A nossa função **output** é provavelmente a mais trivial em todo o nosso projeto: - caso seja dado como argumento um BigNumber positivo, usamos a função concatMap, que percorre a lista de algarismos e concatena-os numa string através da função show - caso o argumento seja um BigNumber negativo, apenas concatena um sinal "-" antes da mesma chamada à função concatMap.

(A partir deste momento vou abreviar BigNumber como BN para simplificar tanto a escrita como a leitura)

Para o cálculo da **soma**, usamos estas pequenas equivalências para nos facilitar os cálculos: apenas caso os dois argumentos sejam positivos o programa vai prosseguir com o cálculo, caso contrário vai adaptar o sinal dos argumentos e/ou do resultado e chamar novamente um possível caso de cálculo.

Por exemplo, no caso da soma: - (+) + (+) = (+) -> Neste caso, o programa vai prosseguir com o cálculo; - (+) + (-) = (+) - (+) -> Caso o 2º argumento seja negativo, é equivalente fazer a subtração de ambos os argumentos positivos; - (-) + (+) = (+) - (+) -> Caso o 1º argumento seja negativo, é equivalente fazer a subtração de ambos os argumentos positivos na ordem inversa; - (-) + (-) -> - ((+) + (+)) -> Caso ambos sejam negativos, é equivalente realizar a soma com ambos positivos e mudar o sinal no final.

A mesma lógica é usada na **subtração** (por exemplo, no caso de 1 BN ser negativo optamos por fazer uma soma dos dois BN's).

Para o caso da **multiplicação**, não há esse problema: apenas precisamos de fazer o cálculo e escolher o sinal correto para o resultado.

Para o verdadeiro cálculo da multiplicação (auxMulBN), começamos por verificar se o segundo argumento é apenas de tamanho 1. Se assim for, chamamos a função carryBN para a multiplicação dos elementos do primeiro argumento com o segundo. Caso contrário, é feita a soma desse mesmo carryBN, com os devidos zeros adicionados pela função *adicionarZerosBN*, com a chamada recursiva a auxMulBN com a restante lista de algarismos do segundo argumento.

Na **divisão**, a estratégia é semelhante às anteriores, também com o caso especial da divisão por zero, que retorna "Exception: Infinity".

Para o verdadeiro cálculo da divisão (auxDivBN), começamos por verificar se o primeiro argumento é maior que o segundo. Se assim for, chamamos novamente a função auxDivBN com o dividendo atualizado (valor de a substraído por b), com o divisor b (mantém-se o mesmo, obviamente) e com o quociente incrementado por 1. Isto é feito recursivamente até que o divisor é maior que o dividendo, retornando o quociente naquele momento e o resto (dividendo naquele momento).

Resposta à alínea 4

Para a resposta a esta alínea, usamos as funções fibListaInfinita e fibListaInfinitaBN para analisar resultados, uma vez que são as mais eficientes em tempos de execução.

(Int -> Int)

4660046610375530309 - fibListaInfinita 92 -> 7540113804746346429 - fibListaInfinita 93 -> -6246583658587674878

(Integer -> Integer)

- fibListaInfinita 92 -> 7540113804746346429
- fibListaInfinita 93 -> 12200160415121876738

(BigNumber -> BigNumber)

- fibListaInfinitaBN (Positive [9,2]) -> Positive [7,5,4,0,1,1,3,8,0,4,7,4,6,3,4,6,4,2,9]
- fibListaInfinitaBN (Positive [9,3]) -> Positive [1,2,2,0,0,1,6,0,4,1,5,1,2,1,8,7,6,7,3,8]

Rapidamente percebemos que algo acontece com Int que não acontece com Integer nem com BigNumber: overflow. Fizemos alguns testes para perceber qual era o limite e rapidamente chegamos aos limites que um Int suporta. Bastou compilar uma função **somar** para, ao correr, recebermos um *warning* sobre os limites caso esse limite fosse excedido:

```
somar :: Int -> Int -> Int
somar a b = a + b
```

- fibListaInfinita 91
  - 4660046610375530309
- fibListaInfinita 92
  - 7540113804746346429
- fibListaInfinita 93
  - -6246583658587674878
- somar 9999999999999999999 0

```
<interactive>:6:7: warning: [-Woverflowed-literals]
    Literal 9999999999999999999 is out of the Int range -9223372036854775808..9223372036854775807
-8446744073709551617
```

Encontrado o problema: o tipo Int em Haskell só aceita valores desde -9223372036854775808 (-2^63) até 9223372036854775807 (2^63 -1).

Com testes mais elaborados e alguma pesquisa, chegamos à conclusão que o tipo Integer não tem qualquer limite como o Int, apenas o limite da memória.

A criação do módulo BigNumber também resolve o problema do limite do Int, uma vez que cada elemento da lista de algarismos tem apenas 1 dígito e que não há qualquer limite para o número de elementos de uma lista.

Concluimos assim que o uso de Integer ou de BigNumber no nosso programa é indiferente (no que toca a comparar resultados, porque o formato é obviamente diferente), uma vez que ambos aceitam inputs de índices extremamente altos sem que ocorra overflow (têm apenas como limite a memória física). O uso de Int só é viável se quisermos trabalhar com valores entre (-2^63, 2^63-1).