Haskell Project

- Documentação e casos de teste
  - Fibonacci
    - FibRec
    - FibLista
    - FibListaInfinita
  - BigNumbers
    - Scanner
    - Output
    - SomaBN
    - SubBN
    - MulBN
    - DivBN
  - Fibonacci com BigNumbers
    - FibRecBN
    - FibListaBN
    - FibListaInfinitaBN
- Estratégias para BigNumbers
  - 1. Tipo
  - 2. Scanner
  - 3. Output
  - 4. somaBN
  - 5. SubBN
  - 6. MulBN
  - 7. DivBN
- Resposta à alínea 4

Documentação e casos de teste

Esta secção segue o seguinte formato:

FunctionName

```
functionName :: type -- id
```

```
testcase 1
testcase 2
testcase 3
...
```

Breve descrição do funcionamento do predicado.

FibRec

```
fibRec :: (Integral a) => a -> a -- 1.1
```

```
fibRec 5
fibRec 10
fibRec 15
```

Parte dos casos base `fibRec 0 = 0` e `fibRec 1 = 1` e recursivamente soma os pares fibonacci `(n-2)` e `(n-1)`.

FibLista

```
fibLista :: Int -> Int -- 1.2
```

```
fibLista 5
fibLista 10
fibLista 15
```

Parte dos casos base `fibRec 0 = 0` e `fibRec 1 = 1` e soma os pares fibonacci `n-2` e `(n-1)`, selecionado (usando `!!`) os elementos `n-2` e `n-1` da lista de fibonacci de `0` até `n`

FibListaInfinita

```
fibListaInfinita :: Int -> Int
```

```
fibListaInfinita 5
fibListaInfinita 10
fibListaInfinita 15
```

Este predicado usa uma função auxiliar `fibInfinitosAux` que cria uma lista infinita de números fibonacci, através do `zipWith (+)` recursivo de duas listas de fibonacci infinitas desfasadas por 1 casa (`lista` e `tail (lista)`). Por terem esse desfasamento, é possível criar uma lista de números fibonacci.

```
0 1 1 2 3 5 (...)

 0 1 1 2 3 5 (...)
```

Scanner

```
scanner :: String -> BigNumber
```

```
scanner "1234"
scanner "-1234"
scanner "0000"
scanner "0001234"
scanner "-0001234"
scanner "12345678901234567890"
```

Percorre todos os caracteres, e transforma-os em inteiros, resultando numa lista de inteiros entre `0` e `9`. Acrescenta o sinal negativo ao primeiro inteiro caso o primeiro char fosse `-`, após ter removido potenciais zeros à esquerda usando uma função auxiliar `removeLeftZeros`.

Output

```
output :: BigNumber -> String
```

```
output [1,2,3,4] -- or output (scanner "1234")
output [-1,2,3,4]
output [0,0,0,0,0]
output [0,0,0,1,2,3,4]
output [1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0]
```

Percorre todos os dígitos de um `BigNumber`, transformando-os em chars e juntando-os numa string, tendo o cuidado de remover potenciais zeros à esquerda.

SomaBN

```
somaBN :: BigNumber -> BigNumber -> BigNumber
```

```
somaBN (scanner "5") (scanner "17")         -- 22
somaBN (scanner "123") (scanner "246")      -- 369
somaBN (scanner "124") (scanner "987")      -- 1111
somaBN (scanner "43632") (scanner "83729") -- 127361
somaBN (scanner "99929834363") (scanner "98934383729") -- 198864218092

somaBN (scanner "123") (scanner "-34")      -- 89
somaBN (scanner "123") (scanner "-130")     -- -7
somaBN (scanner "23") (scanner "-124")      -- -101
```

```
somaBN (scanner "-123") (scanner "33")     -- -90
somaBN (scanner "-123") (scanner "133")    -- 10
somaBN (scanner "-123") (scanner "124")    -- 1

somaBN (scanner "-123") (scanner "-33")     -- -156
somaBN (scanner "-123") (scanner "-133")    -- -256
somaBN (scanner "-9371") (scanner "-29358") -- -38729
```

Soma dois `BigNumbers`. Caso os números não sejam ambos positivos, transforma somas de sinais opostos em subtrações e no caso de serem ambos negativos inverte o sinal dos operandos e da soma final. Antes de somar as duas listas com `zipWith (+) bn1 bn2`, acrescenta zeros à esquerda ao número menos comprido, para as listas serem correspondentes. Processa os *carry outs*, subtraindo 10 a um dígito e acrescentar um ao próximo.

SubBN

```
subBN :: BigNumber -> BigNumber -> BigNumber
```

```
subBN (scanner "123") (scanner "123")     -- 0
subBN (scanner "123") (scanner "246")     -- -123
subBN (scanner "123") (scanner "12")      -- 111
subBN (scanner "9873") (scanner "8328")   -- 1545
subBN (scanner "9839289873") (scanner "983488328") -- 8855801545

subBN (scanner "123") (scanner "-33")     -- 156
subBN (scanner "123") (scanner "-133")    -- 256

subBN (scanner "-123") (scanner "44")     -- -167
subBN (scanner "-123") (scanner "144")    -- -267

subBN (scanner "-123") (scanner "-33")     -- -90
subBN (scanner "-123") (scanner "-133")    -- 10
```

Subtrai dois `BigNumbers`. Caso os números não sejam ambos positivos, transforma subtrações de sinais opostos em somas e no caso de serem ambos negativos inverte o sinal dos operandos e da subtração final. Antes de somar as duas listas com `zipWith (+) bn1 bn2`, acrescenta zeros à esquerda ao número menos comprido, para as listas serem correspondentes. Para além disso subtrai o maior número ao menor número. Processa os *carry outs*, subtraindo 10 a um dígito e acrescentar um ao próximo.

MulBN

```
mulBN :: BigNumber -> BigNumber -> BigNumber
```

```
mulBN (scanner "12") (scanner "24")       -- 288
mulBN (scanner "123") (scanner "12")      -- 1476
mulBN (scanner "123") (scanner "123")     -- 15129
mulBN (scanner "123") (scanner "246")     -- 30258
mulBN (scanner "938123") (scanner "99382246") -- 93232770764258

mulBN (scanner "123") (scanner "-33")     -- -4059
mulBN (scanner "123") (scanner "-133")    -- -16359

mulBN (scanner "-123") (scanner "33")     -- -4059
mulBN (scanner "-123") (scanner "133")    -- -16359

mulBN (scanner "-123") (scanner "-33")     -- 4059
mulBN (scanner "-123") (scanner "-133")    -- 16359
```

Multiplica 2 BigNumbers. Caso os operandos tenham sinais opostos, inverte-se o sinal do número negativo e também o sinal da operação final e se os operandos forem ambos negativos, trocam-se ambos os sinais dos operandos. Começa por ver qual o maior número e transforma-o acrescentando 0s de acordo com a ordem de cada dígito: `[1, 2, 3]` passa a `[100, 20, 3]`. Esse número transformado será multiplicado por cada um dos dígitos do outro número, acrescentando um 0 em cada parcela, consoante a ordem: `123*45` = `([100,20,30] * 5) ++ []` + `([100,20,30] * 4) ++ [0]`. Mais detalhes sobre este predicado no ponto 6 em secção de estratégias.

DivBN

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

```
divBN (scanner "24") (scanner "12")       -- (2,0)
divBN (scanner "30") (scanner "12")       -- (2,6)
divBN (scanner "144") (scanner "12")      -- (12,0)
divBN (scanner "28385") (scanner "4")     -- (7096,1)
```

Para implementar `divBN` (assumindo números não negativos) a nossa abordagem foi a seguinte:

- Começar por fazer uma verificação dos números recebidos usando o predicado já mencionado `bigger` que retorna o par ordenado `(x,y)`:
  - se `x < y` o resultado será `(0, x - y)`
  - se `x == y` o resultado será `(1, 0)`
  - se `x == y * 10^n` para qualquer `n >= 1`, o resultado será `(n, 0)`
- Depois de feitas estas otimizações, podemos avançar e sabemos que `x > y`. Assim sendo o primeiro passo é fazer um `padding` com 0s à direita ao valor de `y` até ao último valor em que `x > y'`, obtendo um valor `y'`. Por exemplo se `x` for `345` e `y` for `3`, `y'` seria `300`, já que `3000` já excede `345`.

- Seguidamente vamos ver *quantos y' cabem em x*, usando `subBN`. Pegando no caso anterior de apenas cabe 1 (`345 - 300 = 45`). Assim o nosso quociente para já é [1], uma lista que vai ficar à espera dos próximos valores.
- A próxima iteração vai usar `x = 45` (*resto*) `y' = init y'`, de maneira a deixar cair um `0` do fim. Caso `init y' == y` sabemos que estamos na última iteração. Para já essa igualdade é falsa: `30 /= 3`. A seguir verificamos que em `45` só cabe um `30`, pelo que o quociente passa a ser [1,1] e o novo x será `45 - 30*1 = 15`.
- A próxima iteração é a final, já que `init [3,0] == [3] == y`. Resta-nos subtrair `3` a `15` até (exclusivé) que o número seja `< 0`.
- `15 - 3*5 = 0`, logo o nosso quociente passa a ser [1,1] ++ [5], com resto `0`.

Mais detalhes sobre este predicado no ponto 7 na secção de estratégias.

FibRecBN

```
fibRecBN :: BigNumber -> BigNumber
```

```
fibRecBN (scanner "5")
fibRecBN (scanner "10")
fibRecBN (scanner "15")
```

Parte dos casos base `fibRec 0 = 0` e `fibRec 1 = 1` e recursivamente soma os pares fibonacci `(n-2)` e `(n-1)`, usando somaBN.

FibListaBN

```
fibListaBN :: BigNumber -> BigNumber
```

```
fibListaBN (scanner "5")
fibListaBN (scanner "10")
fibListaBN (scanner "15")
```

Usa uma função nova `nthBN` para substituir o operador !! nos BigNumbers e partindo dos casos base `fibListaBN [0] = [0]` e `fibListaBN [1] = [1]`, soma os pares fibonacci `n-2` e `(n-1)`, selecionado com `nthBN` os elementos `n-2` e `n-1` da lista de fibonacci de `0` até infinito, já que para não ser infinito precisaríamos de ter o valor absoluto inteiro do BN `n`.

FibListaInfinitaBN

```
fibListaInfinitaBN :: BigNumber -> BigNumber
```

```
fibListaInfinitaBN (scanner "5")
fibListaInfinitaBN (scanner "10")
fibListaInfinitaBN (scanner "15")
```

Este predicado usa uma função auxiliar `fibInfinitosAuxBN` que cria uma lista infinita de `BigNumbers` fibonacci, através do `zipWith somaBN` recursivo de duas listas de fibonacci infinitas desfasadas por 1 casa (`lista` e `tail (lista)`). Por terem esse desfasamento, é possível criar uma lista de números fibonacci.

```
[0] [1] [1] [2] [3] [5] (...)

   [0] [1] [1] [2] [3] [5] (...)
```

SafeDivBN

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

```
divBN (scanner "144") (scanner "12")      -- (12,0)
divBN (scanner "148") (scanner "12")      -- (12,4)
divBN (scanner "148") (scanner "0")       -- Nothing
```

Executa a divisão quando o divisor não é 0, caso contrário retorna `Nothing`

Estratégias para BigNumbers

1. Tipo

Definir o novo tipo como uma lista de inteiros: `type BigNumber = [Int]`

2. Scanner

Para definir `scanner` recorremos ao `map` e `read` para percorrer todos os caracteres, transformando-os em inteiros, tendo o cuidado de acrescentar o sinal negativo caso o primeiro char fosse `-` e removendo potenciais zeros à esquerda usando uma função auxiliar `removeLeftZeros`.

3. Output

Para definir `output` recorremos a `concatMap show` que percorre e junta num array de chars todos os digitos inteiros da lista passada como argumento, tendo o cuidado de remover potenciais zeros à esquerda, recorrendo à função auxiliar `removeLeftZeros`.

4. somaBN

Para definir `somaBN` começámos por criar predicados auxiliares

- `changeSign` para mudar de sinal.
- `isPositive` e `isNegative` para verificar sinal dos BNs
- `padLeftZeros` que acrescenta zeros à esquerda, já que no caso de "somarmos listas" de tamanho diferente, é preciso acrescentar zeros à esquerda num dos operandos até o comprimento ser igual.

O resultado é obtido usando `zipWith (+) n1 n2`. Tendo obtido o resultado, sem pensar nos `carry outs` das somas, pegamos nesse *BigNumber* e usamos a função `carrySum` para lidar com as casas superiores a 9 e outros casos críticos para o resultado final. A nossa implementação de `somaBN` funciona como uma espécie de "router" de resultado:

- caso `x` e `y` sejam ambos positivos é feita uma soma normal com `sumBNResult`.
- caso `x` tenha sinal diferente de `y`, a soma é transformada em subtração (`subBN`)
- caso ambos `x` e `y` sejam ambos negativos a soma é transformada em `- ((-x) + (-y))`

5. SubBN

Para definir `subBN` usámos uma abordagem semelhante à da `somaBN` com as seguintes diferenças relevantes:

- Usamos `carrySub`, que verifica dígitos do *BigNumber* resultante inferiores a 0 e mais alguns casos críticos para o resultado.
- É usado `zipWith (-)` e algumas subtrações são transformadas em somas, por simplicidade.
- Criamos uma função auxiliar `bigger` que se encarrega de retornar um par dos números ordenados, sabendo assim que número é maior e tornando mais fácil a subtração.

6. MulBN

Para implementar `mulBN` seguimos os seguintes passos.

- Começamos por criar uma função axuiliar, `padMulBN`, que acrescenta aos elementos de um *BigNumber* (invertido) zeros `index` vezes. Assim [1, 2, 3] passa a [100, 20, 3].
- O próximo passo é gerir a lógica da multiplicação, de maneira apenas pensarmos em multiplicações de números positivos:
  - caso ambos os operandos sejam positivos faz-se uma multiplicação normal.
  - caso os operandos tenham sinais opostos, inverte-se o sinal do número negativo e também o sinal da operação final.
  - caso os operandos sejam ambos negativos, trocam-se ambos os sinais.
- De seguida temos de recorrer ao predicado auxiliar `bigger` que devolve um par ordenado (a,b), isto é `a > b` (exceto se `a == b`). Assim sabemos sobre que número temos de usar `padMulBN`. `123 * 45 = [100,20,30] * 5` + `([100,20,30] * 4) * 10`.
- É necessário somar todas as operações de multiplicar e para isso usamos `somaBN` e também acrescentamos zeros à lista das multiplicações intermédias caso seja preciso. Pegando no exemplo acima: `123 * 45` seria igual a `([100,20,30] * 5) ++ []` + `([100,20,30] * 4) ++ [0]`. Ou seja, acrescentamos `i` zeros dependendo do índice do dígito operando mais pequeno (invertido) em que nos encontramos.

- No final aplicamos a função `carryMul` para concertar os casos em que um elemento da lista ficou superior a 10: resumidamente a casa com excesso fica com o `a mod 10` e o proximo elemento é `b + a div 10`.

7. DivBN

Para implementar `divBN` (assumindo números não negativos) a nossa abordagem foi a seguinte:

- Começar por fazer uma verificação dos números recebidos usando o predicado já mencionado `bigger` que retorna o par ordenado `(x,y)`:
  - se `x < y` o resultado será `(0, x - y)`
  - se `x == y` o resultado será `(1, 0)`
  - se `x == y * 10^n` para qualquer `n >= 1`, o resultado será `(n, 0)`
- Depois de feitas estas otimizações, podemos avançar e sabemos que `x > y`. Assim sendo o primeiro passo é fazer um `padding` com 0s à direita ao valor de `y` até ao último valor em que `x > y'`, obtendo um valor `y'`. Por exemplo se `x` for `345` e `y` for `3`, `y'` seria `300`, já que `3000` já excede `345`.
- Seguidamente vamos ver *quantos y' cabem em x*, usando `subBN`. Pegando no caso anterior de apenas cabe 1 (`345 - 300 = 45`). Assim o nosso quociente para já é [1], uma lista que vai ficar à espera dos próximos valores.
- A próxima iteração vai usar `x = 45` (*resto*) `y' = init y'`, de maneira a deixar cair um `0` do fim. Caso `init y' == y` sabemos que estamos na última iteração. Para já essa igualdade é falsa: `30 /= 3`. A seguir verificamos que em `45` só cabe um `30`, pelo que o quociente passa a ser [1,1] e o novo x será `45 - 30*1 = 15`.
- A próxima iteração é a final, já que `init [3,0] == [3] == y`. Resta-nos subtrair `3` a `15` até (exclusivé) que o número seja `< 0`.
- `15 - 3*5 = 0`, logo o nosso quociente passa a ser [1,1] ++ [5], com resto `0`.

Resposta à alínea 4

> Compare as resoluções das alíneas 1 e 3 com tipos `(Int -> Int)`, `(Integer -> Integer)` e `(BigNumber -> BigNumber)`, comparando a sua aplicação a números grandes e verificando qual o maior número que cada uma aceita como argumento.

`Integer` define uma precisão arbitrária: não importa quão grand é o número, será representado de acordo com o limite de memória da máquina. This means you never have arithmetic overflows. Por outro lado, `Int` representa um inteiro de 64 bytes, isto é representa números no intervalo `[-2^63, 2^63 - 1]` (em algumas arquiteturas pode ser diferente).

```
9223372036854775808 :: Int  -- testing 2^63 as Int
-- resultado: <interactive>:283:1: warning: [-Woverflowed-literals]
```

A sequência de fibonacci cresce a um ritmo exponencial, pelo que rapidamente o tipo `Int` perde a representação correta. Chamando o predicado de 1.3 `fibListaInfinita 93` obtemos o primeiro resultado negativo, já que esse número é superior a `(2^63)-1` e `Int` não é unsigned.

Se criarmos um predicado igual ao auxiliar da alínea 1.3 mas para `Integers` em vez de `Int` podemos inspecionar que Integer não tem limite e os números da sequência de fibonacci continuam a ser calculados corretamente, sem entrar num limite em que o sinal troca.

```
fibInfinitosInteger :: [Integer]
fibInfinitosInteger = 0 : 1 : zipWith (+) (tail fibInfinitosInteger)
fibInfinitosInteger
```

O tipo `BigNumber` surge para contornar o limite do tipo `Int`, representando os números como listas de inteiros entre `0` e `9`. Tal como a função acima, `fibInfinitosAuxBN` não para de calcular os valores corretos, já que o usa memória livremente.

```
fibInfinitosAuxBN :: [BigNumber]
fibInfinitosAuxBN = [0] : [1] : zipWith somaBN fibInfinitosAuxBN (tail
fibInfinitosAuxBN)
```

Concluindo, enquanto o sistema tiver memória os números continuarão a ser calculados corretamente para o tipo `BigNumber` e `Integer`, enquanto que no tipo `Int` de 64 bytes calcula valores incorretos a partir de 93, já que não consegue representar números positivos superiores a `(2^63)-1`