PFL-Trabalho Prático 1

Contextualização e explicação da resolução dos exercícios do Trabalho Prático 1 da unidade curricular de Programação Funcional em Lógica.

1.1. fibRec

Implementação do cálculo recursivo do enésimo número de Fibonacci. Foram definidos dois casos base quando o número é 0 ou 1 e depois simplesmente é realizada a soma recursiva de:

```
fibRec(number - 1) + fibRec(number - 2)
```

até atingirem os casos base.

1.2. fibLista

Implementação do cálculo do enésimo número de fibonacci de uma lista de resultados parciais. Primeiro é realizado o cálculo da lista de resultados parciais (lista x).

Enquanto o comprimento da lista não for maior do que o índice do elemento que queremos calcular ('a')

```
until( \x -> length x > (fromIntegral a))
```

é criada uma lista que começa com os elementos [0,1] e vai acrescentando a essa lista a soma do último elemento da lista atual com o seu penúltimo elemento.

```
x ++ [last x + last (init x)]
```

Exemplo:

```
[0,1] -> [0,1] ++ [1+0] -> [0,1,1] -> [0,1,1] ...
```

1.3. fibListaInfinita

Implementação do cálculo do enésimo número de Fibonacci de uma lista de números de Fibonacci infinita. A lista de números de Fibonacci infinita é calculada criando uma lista com os casos base, 0 e 1 que são pre-appended e depois com recurso à função zipWith que vai combinar as duas listas elemento a elemento usando a função (+) de soma.

Assim assumindo que temos uma lista infinita de Fibonacci numbers, iremos ter a lista:

```
[ 1, 1, 2, 3,  5,  8, 13, .... ]
```

cuja tail é:

```
[ 1, 2, 3, 5,  8, 13, 21, .... ]
```

resultando em ...

```
[ 1, 1, 2, 3,  5,  8, 13, .... ]
+  [ 1, 2, 3, 5,  8, 13, 21, .... ]
=  [ 2, 3, 5, 8, 13, 21, 34, .... ]
```

No fim para conseguirmos obter o elemento de índice n, apenas temos de aplicar (!!) à lista infinita.

```
fibInf !! (fromIntegral n)
```

2.1. BigNumber

Implementação do type BigNumber, constituído por um Bool e uma lista de Int's. O Bool guarda o sinal do número, True se for positivo e False se for negativo, e a lista guarda os dígitos do número.

2.2. scanner

Esta função converte uma string num BigNumber. Aplica a função read após aplicar a função (:"") que torna cada char numa string, a cada um dos elementos da string str (lista de chars) criando listas individuais de cada um dos números e depois aplica map para criar uma lista de todos os dígitos de str.

Casos de teste

```
*Main> scanner "1234"
(True,[1,2,3,4])
*Main> scanner "-219902"
(False,[2,1,9,9,0,2])
*Main> scanner "0"
(True,[0])
*Main> scanner ""
(True,[])
*Main>
```

2.3. output

Esta função converte um BigNumber numa string. Aplica a função show que converte um elemento lista de dígitos numa string a todos os elementos da lista de dígitos de bigNum através da função map e no fim utiliza a função concat para juntar todos as strings.

Casos de teste

```
*Main> output (True, [1,2,3])
"123"
*Main> output (False, [2,9,0,9,0])
"-29090"
*Main> output (True, [])
""
*Main>
```

2.4. somaBN

Esta função tem como objetivo a soma de dois BigNumbers.

Para ajudar na realização desta soma são utilizadas duas funções auxiliares. Estas funções recebem como argumentos duas listas de dígitos que pertencem correspondentemente a cada um dos BigNumbers e um Int que é designado por carry (número que deve ser passado para a próxima iteração da soma se a soma de dois números for superior a 10), sendo inicialmente igual a 0.

```
6 + 8  = 14 => 4 , carry = 1
```

```
6 + 3 = 9 => 9 , carry = 0
```

A primeira função auxiliar sumBefore realiza a soma das listas de dígitos de dois BigNumbers que tenham sinais iguais. Estas listas são passadas como argumento, recorrendo ao uso da função reverse pois a soma de duas listas dos dígitos de um número torna-se mais fácil quando estão ordenados pela ordem inversa.

123 + 49 = 172 é representado por:

```
[3, 2, 1] + [9, 4] = [2, 7, 1]
```

Esta função soma as duas listas recorrendo a um algoritmo bastante semelhante à soma de dois números manualmente. Soma os dois números correspondentes de cada lista mais o carry e chama recursivamente a função até serem atingidos os casos base (uma das listas, ou as duas ficam vazias).

A segunda função auxiliar subBefore ajuda na soma de dois BigNumbers que tenham sinais opostos pois a soma de dois números de sinais opostos é a subtração do menor número ao maior, sendo preservado o sinal do número que for maior. Para verificar qual número é superior é utilizada outra função auxiliar checkBiggestNum.

O algoritmo utilizado nesta função é o mesmo que na sumBefore apenas sendo diferente no que toca ao cálculo do dígito resultante e do carry.

```
2 - 4 = -2 => 8 , carry = 1
```

```
5 - 4 = 1 => 1  , carry = 0
```

Casos de teste:

```
ghci> somaBN (True, [5,4,3]) (True, [1,2,8])
(True,[6,7,1])
ghci> somaBN (True, [5,4,3]) (False, [1,2,8])
(True,[4,1,5])
ghci> somaBN (False, [5,4,3]) (True, [1,2,8])
(False,[4,1,5])
ghci> somaBN (False, [5,4,3]) (False, [1,2,8])
(False,[6,7,1])
ghci> somaBN (True, [5,4,3,4,5,3,2]) (True, [9,2,4,4,9,9])
(True,[6,3,5,9,0,3,1])
ghci> somaBN (True, []) (True, [])
(True,[])
ghci> somaBN (True, [0]) (True, [1,2,4,5])
(True,[1,2,4,5])
```

2.5. subBN

Esta função tem como objetivo a subtração de dois BigNumbers.

Nesta função são utilizadas da mesma forma todas as funções auxiliares que foram utilizadas na função somaBN, variando apenas os argumentos com que são chamadas.

Casos de teste:

```
ghci> subBN (True, [5,4,3]) (True, [1,2,8])
(True,[4,1,5])
ghci> subBN (True, [5,4,3]) (False, [1,2,8])
(True,[6,7,1])
ghci> subBN (False, [5,4,3]) (True, [1,2,8])
(False,[6,7,1])
ghci> subBN (False, [5,4,3]) (False, [1,2,8])
(False,[4,1,5])
ghci> subBN (True, [5,4,3,4,5,3,2]) (True, [9,2,4,4,9,9])
(True,[4,5,1,0,0,3,3])
ghci> subBN (True, []) (True, [])
(True,[])
ghci> subBN (True, [0]) (True, [1,2,4,5])
(False,[1,2,4,5])
```

2.6. mulBN

Esta função multiplica 2 BigNumbers chamando a função auxiliar "mulAux" com os mesmos argumentos da "mulBN", exceto os sinais que são considerados como positivos para facilitar o cálculo. A função "mulAux"

aplica um algoritmo semelhante à multiplicação manual de forma recursiva. O argumento "BigNum2" é multiplicado por cada dígito do argumento "BigNum1", começando pelo menos significativo, sendo que em cada iteração é adicionado um zero à direita do "BigNum2" e retirado do "BigNum1" o elemento menos significativo. O caso base da função é quando o "BigNum1" têm uma lista vazia. Todas as iterações são somadas para ser obtido o resultado final. O sinal do resultado final é obtido através de um "AND" lógico dos sinais dos argumentos.

Casos de teste:

```
*BigNumber> mulBN (True, [3,2,1]) (True, [1,2,3])
(True,[3,9,4,8,3])
*BigNumber> mulBN (False, [2,0,1,9]) (True, [1,9,9,9])
(False,[4,0,3,5,9,8,1])
*BigNumber> mulBN (True, [2,0,1,9]) (False, [1,9,9,9])
(False,[4,0,3,5,9,8,1])
*BigNumber> mulBN (False, [2,9,1,7,0,1,9]) (False, [9,9,9,9,9,9,9])
(True,[2,9,1,7,0,1,8,7,0,8,2,9,8,1])
*BigNumber>
```

2.7. divBN

Esta função divide 2 BigNumbers chamando a função auxiliar "divAux" com os mesmos argumentos da "divBN", exceto os sinais que são considerados como positivos para facilitar o cálculo. O "divAux" é uma função recursiva que em cada iteração subtrai ao argumento "BigNum1" (numerador) o "BigNum2" (denominador), parando quando o numerador for menor que o denominador. O número de iterações é somado para obter o quociente, consequentemente o resto é obtido através da fórmula:

```
resto = numerador - denominador * quociente
```

O sinal do resultado final é obtido através de um "AND" lógico dos sinais dos argumentos.

Casos de teste:

```
*BigNumber> divBN (True, [3,2,1]) (True, [1,2,3])
((True,[2]),(True,[7,5]))
*BigNumber> divBN (True, [1,2,3]) (True, [3,2,1])
((True,[0]),(True,[1,2,3]))
*BigNumber> divBN (False, [4,1,2,3]) (True, [3,2,1])
((False,[1,2]),(False,[2,7,1]))
*BigNumber> divBN (True, [4,1,2,3]) (False, [3,2,1])
((False,[1,2]),(True,[2,7,1]))
*BigNumber> divBN (False, [4,1,2,3]) (False, [3,2,1])
((True,[1,2]),(False,[2,7,1]))
```

3.1. fibRecBN

Implementação do cálculo recursivo do enésimo número de Fibonacci utilizando BigNumbers.

A implementação é exatamente igual à função normal, porém todas as operações aritméticas realizadas e a sintaxe foram adaptadas de forma a serem compatíveis com os BigNumbers.

```
+ -> somaBN

- -> subBN

1 -> (True, [1])

    etc...
```

3.2. fibListaBN

Implementação do cálculo do enésimo número de fibonacci de uma lista de resultados parciais utilizando BigNumbers.

A implementação é exatamente igual à função normal, porém todas as operações aritméticas realizadas e a sintaxe foram adaptadas de forma a serem compatíveis com os BigNumbers.

```
+ -> somaBN

- -> subBN

1 -> (True, [1])

    etc...
```

3.3. fibListaInfinitaBN

Implementação do cálculo do enésimo número de Fibonacci de uma lista de números de Fibonacci infinita.

A implementação é exatamente igual à função normal, porém todas as operações aritméticas realizadas e a sintaxe foram adaptadas de forma a serem compatíveis com os BigNumbers.

```
+ -> somaBN

- -> subBN

1 -> (True, [1])

    etc...
```

---

4. Comparação

Os tipos **Int**, **Integer** e **BigNumber** variam principalmente na quantidade de número que conseguem representam.

Assim, podemos distinguir os **Int** dos **Integer** pois ao contrário dos **Integer** que podem representar números arbitrariamente grandes, permitindo a representação de números tão grandes quanto a memória do dispositivo permitir, os **Int** têm um limite definido.

Este limite no standard da linguagem é igual a:

```
-2 ^ {29} to (2 ^ {29} - 1)
```

No que toca aos **BigNumbers**, estes à semelhança dos **Integer** também podem representar números arbitrariamente grandes, sendo apenas restringidos pela memória do dispositivo pois representam números através de listas dos seus dígitos e listas podem ser infinitas (não têm limites de tamanho na sua representação).

Apesar dos **Int** conseguirem efetuar operações muito mais rápidas do que os **Integer** ou **BigNumber**, esta restrição de terem um limite atribuído pode causar problemas de overflow ou underflow, levando a bugs indesejados.

---

5. safeDivBN

Esta função divide 2 BigNumbers, porém deteta e impede em compile-time que o denominador seja 0.

Casos de teste:

```
*BigNumber> safeDivBN (True, [4,1,2,3]) (True, [0])
Nothing
*BigNumber> safeDivBN (True, [4,1,2,3]) (False, [3,2,1])
Just ((False,[1,2]),(True,[2,7,1]))
*BigNumber>
```

Grupo 506, 28/11/2021

- João Andrade, up201905589@up.pt
- Sérgio Estêvão, up201905680@up.pt