PFL_TP1_G7_05

Configuração e/ou Instalação

Neste caso, não são necessárias nenhumas configurações extraordinárias, bastando apenas entrar no interpretador ***ghci*** e carregar os ficheiros **Fib.hs** e **BigNumber.hs**, em concordância com as funções que pretende utilizar.

Estas instruções são válidas tanto em *Windows* como em *Linux*.

Explicação de funções

Por cada ficheiro de código-fonte é apresentado, por função, uma explicação sucinta do seu funcionamento e uma descrição de vários casos de teste.

Em específico, para as funções da alínea 2 também são exploradas as estratégias utilizadas na sua implementação.

1 - Fib.hs

`fibRec`

Funcionamento: Implementa uma abordagem recursiva no cálculo do enésimo número de Fibonacci.

Casos de teste:	
Índice 0	fibRec 0 = 0
Índice 1	fibRec 1 = 1
Índice 10	fibRec 10 = 55

`somaUltimos2`

Funcionamento: Soma os últimos dois elementos de uma lista: último e penúltimo. Assume que a lista tem no minimo dois elementos.

Casos de teste:	
Lista com 2 elementos	somaUltimos2 [0,1] = 1
Lista com mais que 2 elementos	somaUltimos2 [0,1,1,2,3] = 5

`expandirSeq`

Funcionamento: Aumenta recursivamente uma determinada lista em 'n' elementos, sendo que um novo elemento é calculando a partir da soma dos dois ultimos.

Casos de teste:	
Aumentar a lista em 0 elementos	expandirSeq [0,1] 0 = [0,1]
Aumentar a lista em 1 elemento	expandirSeq [0,1] 1 = [0,1,1]
Aumentar a lista em mais que 1 elemento	expandirSeq [0,1] 4 = [0,1,1,2,3,5]

`fibLista`

Funcionamento: Implementa uma abordagem de programação dinâmica no cálculo do enésimo número de Fibonacci. Tira partido de uma lista auxiliar com resultados parciais.

Casos de teste:	
Índice 0	fibLista 0 = 0
Índice 1	fibLista 1 = 1
Índice 10	fibLista 10 = 55

`fibListaInfinita`

Funcionamento: Implementa uma abordagem que calcula o enésimo número de Fibonacci apartir da geração de uma lista infinita.

Casos de teste:	
Índice 0	fibListaInfinita 0 = 0
Índice 1	fibListaInfinita 1 = 1
Índice 10	fibListaInfinita 10 = 55

`fibRecBN`, `somaUltimos2BN`, `expandirSeqBN`, `fibListaBN`, `fibListaInfinitaBN`

Aplicação da mesma funcionalidade descrita nos pontos acima, mas para o tipo BigNumber. Os casos de teste aplicados aqui foram os mesmos apresentados nos pontos anteriores adaptados de forma a cumprirem a sintaxe inerente a este tipo de dados.

2 - BigNumber.hs

`zip0`

Funcionamento: Os elementos de duas listas são emparelhados, formando pares, e quando uma delas termina a outra continua a ser emparelhada com o *Int* 0 até não ter mais elementos.

Casos de teste:	
Emparelhamento de listas vazias	zip0 [] [] = []
Emparelhamento de listas de igual tamanho	zip0 [1,2,3] [3,2,1] = [(1,3),(2,2),(3,1)]
Emparelhamento de listas de diferente tamanho	zip0 [1,2] [3] = [(1,3),(2,0)]; zip0 [3] [1,2] = [(3,1),(0,2)]

`cleanLeft0s`

Funcionamento: Dada uma lista de *Int*, são eliminados desta os primeiros elementos que forem o *Int* 0. É assegurado que fica pelo menos um elemento na lista permanecendo sempre o último elemento da mesma, independentemente do seu valor.

Casos de teste:	
Lista com apenas 0's	cleanLeft0s [0] = [0]; cleanLeft0s [0,0,0] = 0
Lista sem zeros à esquerda	cleanLeft0s [1] = [1]; cleanLeft0s [1,0,0] = [1,0,0]
Lista com zeros à esquerda	cleanLeft0s [0,1] = [1]; cleanLeft0s [0,0,1,0] = [1,0]

`bnList`

Funcionamento: Retorna a lista de dígitos de um BigNumber, ignorando o seu sinal.

Casos de teste:	
BigNumber Positivo	(Positive [1,2,3]) = [1,2,3]
BigNumber Negativo	(Negative [1,2,3]) = [1,2,3]

`stringToN`

Funcionamento: Percorre a lista de *Char* e cria uma lista de *Int* aplicando a função *digitToInt* do módulo *Data.Char*, transformando assim cada dígito da *String* no seu inteiro respetivo e adicionando este à lista de inteiros.

Casos de teste:	
String vazia	stringToN "" = []
String constituida apenas por digitos	stringToN "123" = [1,2,3]

`scanner`

Funcionamento/Estratégia: Inicializa o *BigNumber* tirando partido da lista de *Int* gerada pela função *stringToN* referida anteriormente e limpando os dígitos não significativos pela aplicação da função *cleanLeft0s*. Tem em conta o primeiro caracter da *String* de forma a distinguir o *BigNumber* como positivo (*Positive*) ou negativo (*Negative*). Na ausência desse caracter de sinal é assumido que o número representado por essa *String* é positivo.

Casos de teste:	
String de um número negativo	scanner "-123" = Negative [1,2,3]
String de um número positivo com sinal	scanner "+123" = Positive [1,2,3]
String de um número positivo sem sinal	scanner "123" = Positive [1,2,3]
String de um número positivo com zeros à esquerda	scanner "0123" = [1,2,3]

Nota: assumimos que a *String* recebida como argumento é válida e não tem mais nada para além do sinal seguido de digitos.

`nToString`

Funcionamento: Percorre uma lista de *Int* adicionando ao resultado a representação em *String* de cada um dos elementos pela ordem em que se encontram na lista, sem descartar nenhum. Para obter a *String* de um *Int* é aplicada a função *show*.

Casos de teste:	
Lista vazia	nToString [] = ""
Lista composta por digitos	nToString [0,1,2,3] = "0123"

`output`

Funcionamento/Estratégia: Converte um *BigNumber* em *String*, na qual o primeiro caracter representa o sinal ('+' ou '-') seguido dos dígitos que constituem o número representado pelo *BigNumber*, descartando possiveis 0's que possam existir à esquerda do valor de um número, com o auxílio das funções *nToString* e *cleanLeft0s*.

Casos de teste:	
BigNumber Negativo	output (Negative [2,3]) = "-23"
BigNumber Positivo	output (Positive [2,3]) = "+23"
BigNumber Positivo com zeros à esquerda	output (Positive [0,2,0]) = "+20"

`sumDigits`

Funcionamento: Aplica o algoritmo da soma da primária (contas em pé), onde somamos cada par de dígitos guardando o dígito das unidades e passando para a próxima soma o resto que resulta dessa adição. A soma é realizada da esquerda para a direita do conjunto de pares (primeiro par corresponde aos algarismos das unidades).

Casos de teste:	
Um par sem resto	sumDigits [(1,2)] = [3]
Um par com resto	sumDigits [(1,9)] = [0,1]
Vários pares de dígitos a somar	sumDigits [(1,2), (3,9)] = [3,2,1]

`somaBN`

Funcionamento: Soma dois *BigNumber*, apresentando o resultado sem zeros à esquerda.

Estratégia: Os dígitos dos dois *BigNumber* a somar são emparelhados de forma a que em cada par de dígitos se encontrem os dígitos do mesmo índice (unidades, dezenas, centenas, etc) e a que este conjunto de pares contenha mais à esquerda os digitos de menor peso no valor do número. De seguida é aplicada a função *sumDigits* referida anteriormente para efetuar o cálculo. O resultado equivale assim a um *BigNumber* cujos dígitos correspondem à soma destes pares em ordem reversa. Exemplificando:

```
resto:                1         0         0
          123         123       123       123
        + 359  ->   + 359  ->  + 359  ->  + 359
        ------      ------    ------    ------
                      2         82        482
```

Casos de teste:	
Soma de zeros	somaBN (Positive [0]) (Positive [0]) = Positive [0]
Soma de negativos	somaBN (Negative [1,2,3]) (Negative [3,5,9]) = Negative [4,8,2]
Soma de positivos	somaBN (Positive [1,2,3]) (Positive [3,5,9]) = Positive [4,8,2]

Nota: As somas entre um número negativo e positivo foram traduzidas numa subtração tomando partido da propriedade comutativa destas operações aritméticas. A subtração é realizada com a função *subBN* referida posteriormente.

`subDigits`

Funcionamento: Aplica o algoritmo da subtração da primária (contas em pé), onde subtraimos cada par de dígitos guardando o dígito das unidades e adicionando ao dígito do elemento de baixo o resto que resulta dessa subtração sempre que em cada par o primeiro dígito é menor do que o segundo. A subtração é realizada da esquerda para a direita da lista de pares (primeiro par corresponde aos algarismos das unidades). Assume-se que o conjunto de primeiros elementos dos pares representam um número superior, em módulo, ao número representado pelo conjunto de segundos elementos dos pares.

Casos de teste:	
Um par	subDigits [(5,1)] = [4]
Pares sem resto a transportar	subDigits [(5,3), (2,2)] = [2,0]
Pares com resto a transportar	subDigits [(2,3), (4,2)] = [9,1]

`subBN`

Funcionamento: Subtrai dois BigNumber, apresentando o resultado sem zeros à esquerda.

Estratégia: Para subtrair dois positivos o sinal do resultado corresponde ao sinal daquele da parcela que é maior em valor, enquanto o valor corresponde a subtrair à maior parcela a menor. Para obter este último valor subtrai-se cada par de digitos que corresponde ao mesmo indice, começando pelas unidades, tratando corretamente o resto e apresentando o resultado sem zeros à esquerda.

```
                      (6 para 12) (1 passa a 2)
   420        420        420        420
 - 160  ->  - 160  ->  - 160  ->  - 260
------     ------     ------     ------
                0         60        260
```

Casos de teste:	
Subtrair por zero	subBN (Positive [1,2]) (Positive [0]) = Positive [1,2]
Subtrair numeros iguais	subBN (Positive [1,2]) (Positive [1,2]) = Positive [0]
Subtrair com transporte de resto	subBN (Positive [4,2,0]) (Positive [1,6,0]) = Positive [2,6,0]

Nota: As subtrações entre um número negativo e positivo foram traduzidas numa soma tomando partido da propriedade comutativa destas operações aritméticas, bem como a subtração entre dois negativos é traduzida na subtração de dois positivos.

`mulDigit`

Funcionamento: Multiplica um número (representado pela sua lista de dígitos) por um dígito, transportando o resto que resulta de cada multiplicação individual para a multiplicação seguinte. O dígito mais à esquerda corresponde às unidades.

Casos de teste:	
Multiplicação por 0	mulDigit [2,1] 0 0 = [0]
Multiplicação por 1	mulDigit [2,1] 1 0 = [2,1]
Multiplicação sem transporte	mulDigit [2,1] 2 0 = [4,2]
Multiplicação com transporte	mulDigit [2,1] 6 0 = [2,7]

Nota: Como a multiplicação e a adição não são comutativas também é necessário iniciar o resto a transportar como argumento da função.

`mulParcelas`

Funcionamento: Forma todas as parcelas que resultam da multiplicação da primeira lista de dígitos por cada um dos dígitos da segunda lista, sendo que cada parcela é deslocada (colocados zeros à esquerda) de forma a que o primeiro dígito não acrescentado de cada resultado intermédio corresponda ao índice do dígito pelo qual foi multiplicado na segunda lista. O dígito da esquerda corresponde ao das unidades.

Casos de teste:	
Segunda lista com 1 digito	mulParcelas [2,1] [2] = [[4,2]]
Segunda lista com 2 digitos	mulParcelas [2,1] [2,1] = [[4,2],[0,2,1]]

`mulNs`

Funcionamento: Multiplica duas listas de dígitos somando todas as parcelas que resultam da multiplicação da primeira lista por cada um dos dígitos da segunda (a lista de parcelas a serem somadas é gerada pela função *mulParcelas* referida anteriormente). Recebe e retorna a lista de dígitos que representa os numeros pela ordem de leitura (à esquerda o de maior valor).

Casos de teste:	
Multiplicação sem transporte por 1 dígito	mulNs [1,2] [2] = [2,4]
Multiplicação sem transporte por mais dígitos	mulNs [1,2] [1,2] = [1,4,4]
Multiplicação com transporte por 1 dígito	mulNs [1,2] [6] = [7,2]
Multiplicação com transporte por mais dígitos	mulNs [1,2] [6,0] = [7,2,0]

`mulBN`

Funcionamento: Aplica as propriedades da multiplicação no que toca aos sinais, fazendo corresponder o sinal correto à multiplicação obtida que foi gerada independentemente dos sinais dos argumentos.

Estratégia: Multiplicação da primária (conta em pé)

```
            (4*12)     (3*12)     (somar)
   12         12         12         12
 * 34  ->   * 34  ->   * 34  ->   * 34
------     ------     ------     ------
             48         48         48
                       360      + 360
                                ------
                                  408
```

Casos de teste:	
Multiplicação de negativos	mulBN (Negative [1,2]) (Negative [1,2]) = Positive [1,4,4]
Multiplicação de positivos	mulBN (Positive [1,2]) (Positive [1,2]) = Positive [1,4,4]
Multiplicação de positivo por negativo	mulBN (Negative [1,2]) (Positive [1,2]) = Negative [1,4,4]

`sucBN`

Funcionamento: Produz uma lista infinita de *BigNumber*, cujo primeiro elemento é o argumento *i* e os seguintes são o resultado de, recursivamente, somar 1 unidade ao elemento anterior na lista.

Casos de teste:	
Sequência de numeros inteiros maiores e iguais a zero em *BigNumber*	take 10 (sucBN (Positive [0])) = [Positive [0],Positive [1],Positive [2],Positive [3],Positive [4],Positive [5],Positive [6],Positive [7],Positive [8],Positive [9]]

`divBN`

Funcionamento: Efetua a divisão inteira de dois *BigNumber*, retornando um par "(quociente, resto)". Assume que ambos os argumentos são positivos.

Estratégia: Começa por produzir uma lista de pares *(q, dq)* em que *dq* corresponde a múltiplos do divisor e *q* corresponde ao multiplicador que dá origem a *dq* (ou seja, *divisor * q = dq*). Desta lista infinita de pares ordenada, filtra-se apenas os elementos cujo *dq* é menor ou igual que o *dividendo* da divisão. Daí retira-se o *q* e o *dq* do último elemento da lista que correspondem, respetivamente, ao *quociente* e ao valor pelo qual a subtração do *dividendo* por ele resulta no *resto*. Exemplificando:

```
25/4 -> qual o maior múltiplo de 4 que é menor ou igual a 25? R.: 24
     -> qual o número que multiplicado por 4 dá 24? R.: 6 = quociente
     -> qual o resto? R.: 25 - 24 = 1
Lista gerada:
[(0,0), (1,4), (2,8), (3,12), (4,16), (5,20), **(6,24)**, (7,28), ...]
```

Casos de teste:	
Divisão com quociente 0	divBN (Positive [4]) (Positive [5]) = (Positive [0],Positive [4])
Divisão por um múltiplo	divBN (Positive [1,2]) (Positive [4]) = (Positive [3],Positive [0])
Divisão com resto diferente de 0	divBN (Positive [2,5]) (Positive [4]) = (Positive [6],Positive [1])

`safeDivBN`

Funcionamento/Estratégia: Deteta divisões por zero em compile-time. Como tal, retorna monads do tipo Maybe. Pelo que só efetua uma divisão se e só se o divisor for diferente de 0.

Casos de teste:	
Divisão por 0	safeDivBN (Positive [4]) (Positive [0]) = Nothing
Divisão com divisor diferente de 0	safeDivBN (Positive [4]) (Positive [3]) = Just (Positive [1],Positive [1])

Resposta à alínea 4

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

(Int -> Int)

O tipo *Int* é um tipo de precisão fixa. Permite representar inteiros de 64 bits, pelo que o maior número possível de ser representado corresponde a *2^63 - 1 = 9 223 372 036 854 775 807*. Tal informação foi também verificada pelo recurso à função **maxBound**:

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

Figura 1: Limites de Representação do tipo Int

Testando assim uma versão das alíneas da pergunta 1, mas restringindo o tipo ao pretendido, obtivemos os seguintes resultados:

Argumento (n)	Resultado Esperado	Resultado Obtido
90	2 880 067 194 370 816 120	2 880 067 194 370 816 120
91	4 660 046 610 375 530 309	4 660 046 610 375 530 309
92	7 540 113 804 746 346 429	7 540 113 804 746 346 429
93	12 200 160 415 121 876 738	-6 246 583 658 587 674 878

Concluimos, que o maior número que é aceite como argumento é **92**, último cujo valor na sequência é menor ao limite de *INT*. Como é evidente na tabela a cima, com o argumento 93 o valor de retorno sofre *overflow*, não devolvendo o valor esperado (*12 200 160 415 121 876 738*).

(Integer -> Integer)

O tipo *Integer* é um tipo de precisão arbitrária, isto significa que consegue representar qualquer número independentemente do seu tamanho desde que haja memória suficiente. O limite de representação é assim tão maior quanto a memória que existe disponível. Isto traduz-se na inexistência de *overflows*, no entanto, as operações aritméticas tornam-se relativamente lentas. Normalmente, é um tipo de dados usado quando se considera que um *overflow* com *Int* poderá acontecer, compromentendo a eficiência.

```
ghci> (minBound, maxBound) :: (Integer, Integer)

<interactive>:44:2: error:
    * No instance for (Bounded Integer)
        arising from a use of `minBound'
    * In the expression: minBound
      In the expression: (minBound, maxBound) :: (Integer, Integer)
      In an equation for `it':
          it = (minBound, maxBound) :: (Integer, Integer)
```

Figura 2: Limites de Representação do tipo Integer

Listagem de algumas chamadas realizadas à função, para teste:

- fibListaInfinita 95
- fibListaInfinita 1000
- fibListaInfinita 930000

Após estes testes, onde os valores na sequência que correspondem a estes índices são número extramamente grandes, concluímos que não conseguimos apresentar o maior número que é aceite como argumento, uma vez que tal está diretamente relacionado com a memória disponível.

Assim, a partir destes testes assim como os testes para os outros tipos, concluimos que o *Integer* é o tipo ideal a usar quando pretendemos representar números relativamente grandes, uma vez que tem uma representação pseudo-infinita, assim como uma eficência nas operações aritméticas superior à do tipo *BigNumber*.

(BigNumber -> BigNumber)

O tipo *BigNumber* é um tipo que representa um número pela lista dos seus dígitos. Por isso, não existe um limite de representação tão rígido como explorado nos pontos acima. Apesar de também acabar por estar limitado pela memória disponível, por se tratar de uma lista, os seus diferentes elementos - tendo em conta a forma como o *Haskell* a organiza - não necessitam de estar consecutivos em memória, tornando a sua alocação mais eficiente. Por outro lado, podemos também realçar que na definição deste tipo de dados nenhuma operação aritmética inerente ao cálculo da sequência de fibonacci gera *overfow*, já que estas apenas são feitas entre dígitos e/ou inteiros muito pequenos. Por outro lado, não se mostraram tão eficientes como o tipo *Integer* na obtenção do número de Fibonacci de ordem *n* uma vez que atige o máximo da memória disponível em índices inferiores.

Listagem de algumas chamadas realizadas à função, para teste:

- fibListaInfinitaBN 95
- fibListaInfinitaBN 1000
- fibListaInfinitaBN 93000

Tal como no ponto anterior, não é possível precisar o maior número que consegue receber como argumento.

Diferentes Abordagens e os Tipos

A complexidade temporal e espacial das 3 formas de implementação da sequência de fibonacci também influencia de igual forma o limite máximo do número a representar, independentemente do tipo de dados a manipular.

1. As funções menos eficientes são as de abordagem recursiva exigentes a nível, tanto de espaço como de tempo.
2. No intermédio, estão as funções de abordagem de programação dinâmica muito exigentes a nivel de espaço o que compromete e influencia o espaço na memória disponível para a representação de cada um dos elementos da lista auxiliar. (fibListaBN 930000 => ocupação de memória de 97% no limite)
3. As funções de abordagem de lista infinita são as mais eficientes, tanto a nível de espaço como de tempo, permitindo assim reservar mais espaço para os números a representar, recebidos tanto como argumento como aqueles computacionados.

Henrique Nunes (up201906852)

Patrícia Oliveira (up201905427)