
PROJETO PRÁTICO I

PROGRAMAÇÃO FUNCIONAL E EM LÓGICA

João Baltazar, Nuno Costa

201905616, 201906272

LEIC, Faculdade de Engenharia

Universidade do Porto

27 de novembro de 2021

Conteúdo

1	Descrição de Funções	1
1.1	Módulo de Fibonacci	1
1.1.1	fibRec	1
1.1.2	fibLista	1
1.1.3	fibListaInfinita	2
1.2	Módulo de BigNumber	3
1.2.1	Definição do BigNumber	3
1.2.2	Definição da função scanner	3
1.2.3	Definição da função output	3
1.2.4	Definição da função somaBN	4
1.2.5	Definição da função subBN	5
1.2.6	Definição da função mulBN	6
1.2.7	Definição da função divBN	7
1.3	Extensão do Módulo de Fibonacci a BigNumber	8
1.3.1	Definição da função fibRecBN	8
1.3.2	Definição da função fibListaBN	8
1.3.3	Definição da função fibListaInfinitaBN	9
1.4	Função de Divisão Segura	9
2	Casos de Teste	10
2.1	fibRec	10
2.2	fibLista	10
2.3	fibListaInfinita	10
2.4	scanner	10
2.5	output	11
2.6	somaBN	11
2.7	subBN	11
2.8	mulBN	11
2.9	divBN	12
2.10	fibRecBN	12
2.11	fibListaBN	12
2.12	fibListaInfinitaBN	12
2.13	safeDivBN	12
3	Alínea 4 - Análise e Comparação das resoluções das alíneas 1 e 3	13

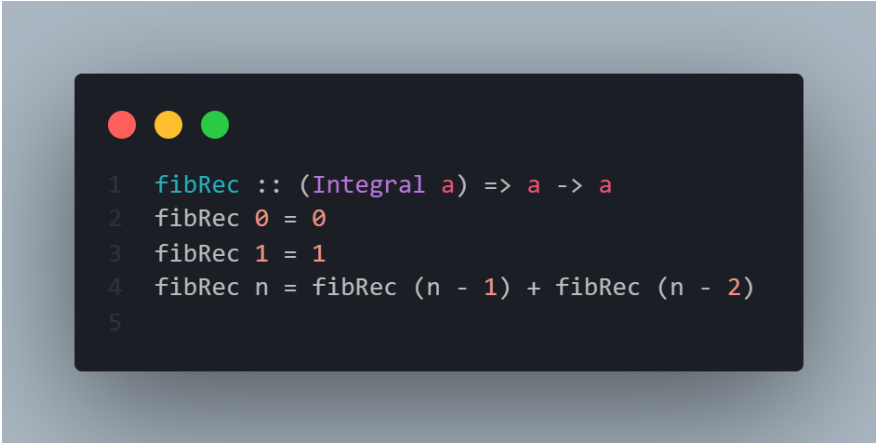
1 Descrição de Funções

1.1 Módulo de Fibonacci

Para este módulo foram criadas funções de tipo genérico, para abranger inputs tanto do tipo *Int* como *Integer*.

1.1.1 fibRec

Figura 1: Definição da função fibRec

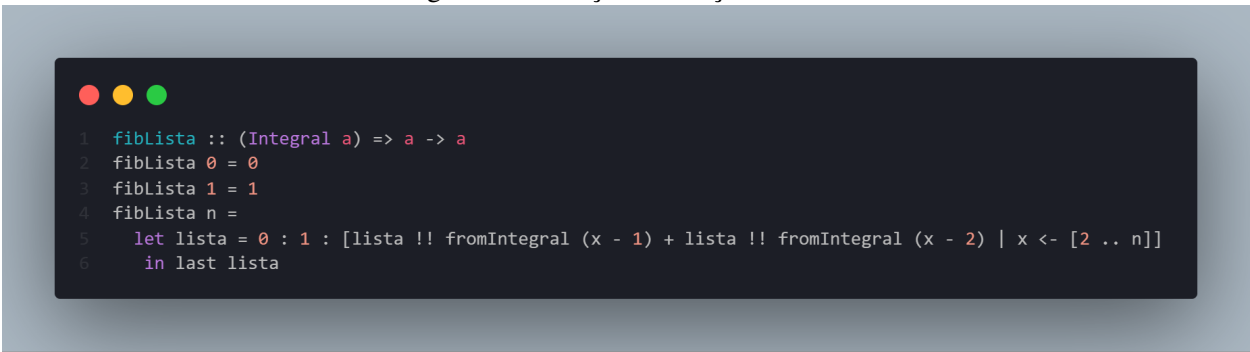


```
1 fibRec :: (Integral a) => a -> a
2 fibRec 0 = 0
3 fibRec 1 = 1
4 fibRec n = fibRec (n - 1) + fibRec (n - 2)
5
```

Definição trivial recursiva da sequência de Fibonacci. Nesta implementação, são definidos os padrões representativos do caso base (0 e 1), e é definida a versão recursiva para os restantes valores - equivalente à soma dos dois valores de índice imediatamente inferior a *n*.

1.1.2 fibLista

Figura 2: Definição da função fibLista

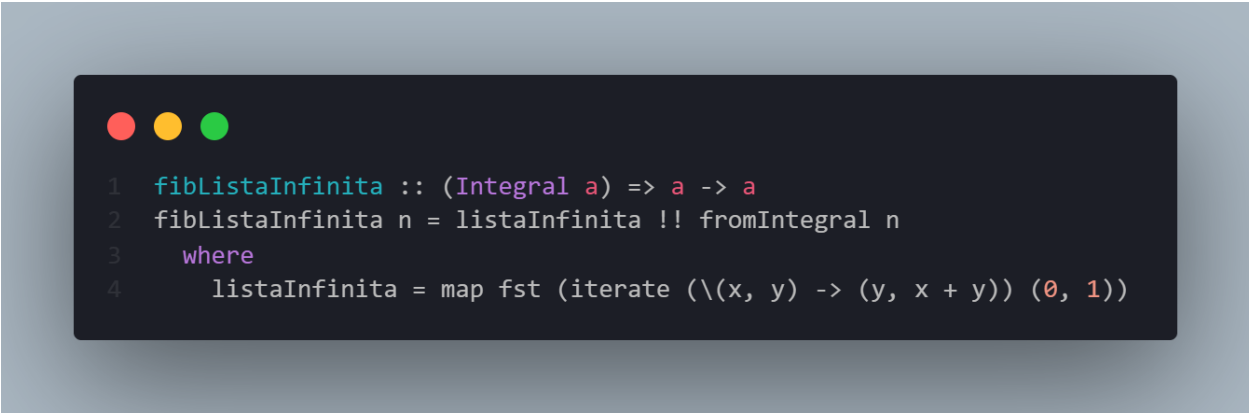


```
1 fibLista :: (Integral a) => a -> a
2 fibLista 0 = 0
3 fibLista 1 = 1
4 fibLista n =
5   let lista = 0 : 1 : [lista !! fromIntegral (x - 1) + lista !! fromIntegral (x - 2) | x <- [2 .. n]]
6   in last lista
```

Definição, em lista, da sequência de Fibonacci. Tal como na implementação recursiva, são definidas duas guardas para os casos base. No entanto, para os restantes valores é definida uma lista, que gera, até ao índice *n*, o conjunto de valores tais que o valor de índice *i* é igual à soma dos valores de índice *i-1* e *i-2* - seguindo, portanto, a regra da sequência de Fibonacci.

1.1.3 fibListaInfinita

Figura 3: Definição da função fibListaInfinita



```
1 fibListaInfinita :: (Integral a) => a -> a
2 fibListaInfinita n = listaInfinita !! fromIntegral n
3   where
4     listaInfinita = map fst (iterate (\(x, y) -> (y, x + y)) (0, 1))
```

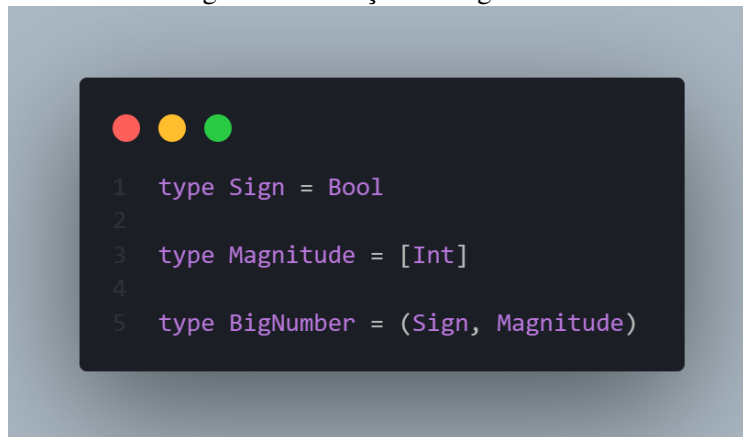
Definição, em lista infinita, da sequência de Fibonacci. É possível utilizar a função *iterate*, que gera, iterativamente, uma lista infinita, aplicando ao último elemento uma função. Gera-se, assim, uma lista com os valores da sequência de Fibonacci no formato [(x,y),(y,z)..], tirando proveito de que para o cálculo desta sequência apenas são necessários os dois valores anteriores. Mapeando a função *fst* a todos os elementos da lista resultante, obtém-se uma lista - infinita - da sequência de Fibonacci. Para o resultado final, é retornado o n-ésimo elemento da lista.

1.2 Módulo de BigNumber

Para este módulo foram criadas funções aritméticas a aplicar em números representados no formato de lista de dígitos - ou BigNumber. Genericamente, procurou-se utilizar os algoritmos básicos de soma, subtração, multiplicação e divisão utilizados manualmente.

1.2.1 Definição do BigNumber

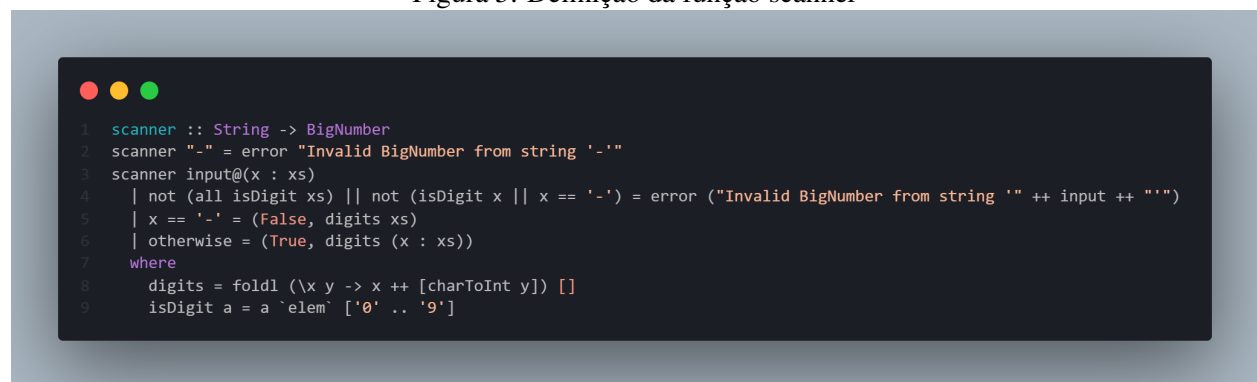
Figura 4: Definição do BigNumber



Foi escolhida esta representação para BigNumber. O primeiro campo - **Sign** - indica o sinal do número - *False* para **Negativo**, *True* para **Positivo**. O segundo campo - **Magnitude** - representa a lista sequencial de dígitos que formam o número.

1.2.2 Definição da função scanner

Figura 5: Definição da função scanner



A função *scanner* gera, para um dado inteiro representado em string de dígitos - com o símbolo '-' para indicar sinal negativo e a sua ausência a indicar sinal positivo - um BigNumber, no formato previamente declarado. É feita a testagem e o tratamento de inputs indevidos.

1.2.3 Definição da função output

A função *output* gera, para um dado BigNumber, a representação inteira em string. Para gerar a string de dígitos, é aplicado um *foldl* ao campo **Magnitude** utilizando a função *show*. É a função inversa da função *scanner*.

Figura 6: Definição da função output

```

1 output :: BigNumber -> String
2 output (_, []) = error "Invalid BigNumber: Empty List"
3 output num
4   | sign num = strNum
5   | otherwise = "-" ++ strNum
6   where
7     strNum = foldl1 (\x y -> x ++ show y) "" (mag num)

```

1.2.4 Definição da função somaBN

Figura 7: Definição da função somaBN

```

1 somaBN :: BigNumber -> BigNumber -> BigNumber
2 somaBN x y
3   | sign x == sign y = rectifyZeroSign (sign x, usoma (mag x) (mag y))
4   | sign x = rectifyZeroSign (subBN x (symmetric y))
5   | otherwise = rectifyZeroSign (subBN y (symmetric x))

```

A função *somaBN* calcula a soma de dois *BigNumbers*. Para tal, são utilizadas as propriedades dos dois números. Caso ambos os números tenham o mesmo sinal, a sua soma apenas se irá refletir na soma das suas magnitudes, mantendo o sinal (esta soma de magnitudes é detalhada abaixo). Caso não tenham, e caso o primeiro número seja positivo - o que indica que o segundo é negativo - esta operação é uma simples subtração do segundo ao primeiro. Caso contrário, é uma subtração do primeiro ao segundo. Tira-se proveito, portanto, da função *subBN*, abaixo detalhada.

Figura 8: Definição da função auxiliar de soma

```

1 usomaaux :: Magnitude -> Magnitude -> Int -> Magnitude
2 usomaaux [] [] c = [c]
3 usomaaux left right c
4   | res > 9 = res `mod` 10 : usomaaux newRight newLeft 1
5   | otherwise = res : usomaaux newRight newLeft 0
6   where
7     newRight = if null right then [] else tail right
8     newLeft = if null left then [] else tail left
9     res
10    | null left && null right = c
11    | null left = head right + c
12    | null right = head left + c
13    | otherwise = head right + head left + c
14
15 usoma :: Magnitude -> Magnitude -> Magnitude
16 usoma x y = rmlleadzeros (reverse (usomaaux (reverse x) (reverse y) 0))

```

Para a soma de magnitudes, é utilizada a lógica de soma primária com campo de *carry*. Assim, soma-se os dígitos por ordem de magnitude, e caso essa soma seja superior à magnitude que representa (no caso da

soma, se for maior que 10), é passado o *carry* correspondente (neste caso 1). Esse *carry* é, então, somado na operação da ordem de magnitude seguinte.

1.2.5 Definição da função *subBN*

Figura 9: Definição da função *subBN*

```

1 subBN :: BigNumber -> BigNumber -> BigNumber
2 subBN x y
3   | sign x /= sign y = rectifyZeroSign (somaBN x (symmetric y))
4   | otherwise = rectifyZeroSign (not (cmpBN x y), usub (mag x) (mag y))

```

A função *subBN* calcula a subtração de dois *BigNumbers*. Para tal, são utilizadas, novamente, as propriedades dos dois números. Caso ambos os números tenham sinal diferente, ao subtrair um ao outro passamos a ter uma soma de dois números com o mesmo sinal. Assim, é chamada a função *somaBN* para tratar da operação. Caso tenham o mesmo sinal, é feita subtração primária das magnitudes dos dois números (explicado em maior detalhe em baixo), e o sinal é o mesmo que o do número de maior magnitude.

Figura 10: Definição da função auxiliar de subtração

```

1 -- assumes x is the largest
2 usubaux :: Magnitude -> Magnitude -> Int -> Magnitude
3 usubaux [] [] 0 = []
4 usubaux x [] c = head x - c : tail x
5 usubaux (x : xs) (y : ys) c
6   | res < 0 = res `mod` 10 : usubaux xs ys 1
7   | otherwise = res : usubaux xs ys 0
8   where
9     res = x - (y + c)
10
11 -- subtracts the smallest from the largest
12 usub :: Magnitude -> Magnitude -> Magnitude
13 usub x y
14   | smaller = rmlleadzeros (reverse (usubaux yr xr 0))
15   | otherwise = rmlleadzeros (reverse (usubaux xr yr 0))
16   where
17     smaller = cmpMag x y
18     xr = reverse x
19     yr = reverse y

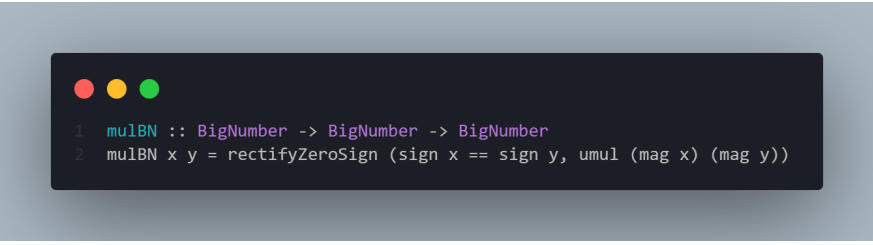
```

Para a subtração de magnitudes, é utilizada a lógica de subtração primária com campo de *carry* - idêntico à soma, ainda que a lógica difira ligeiramente. Assim, subtraem-se os dígitos por ordem de magnitude, e caso essa subtração seja inferior a 0, é passado o *carry* simétrico desse resultado (neste caso 1). Esse *carry* é, então, somado na operação da ordem de magnitude seguinte à magnitude de subtração, tal que a próxima operações seja da forma *subtraído* - (*subtração* + *carry*).

De notar que é utilizada uma função auxiliar extra que assegura que a subtração é feita da mais pequena à maior das magnitudes, dado que a magnitude representa um número inteiro positivo - isto é, não tem em conta o sinal.

1.2.6 Definição da função mulBN

Figura 11: Definição da função mulBN




```

1 mulBN :: BigNumber -> BigNumber -> BigNumber
2 mulBN x y = rectifyZeroSign (sign x == sign y, umul (mag x) (mag y))

```

A função *mulBN* calcula a multiplicação de dois *BigNumbers*. É sabido que se os sinais forem iguais, o sinal do resultado será positivo. Tirando partido dessa propriedade, é possível calcular o sinal do resultado facilmente. A magnitude resultante é obtida através de uma outra função auxiliar (abaixo exposta e explicada).

Figura 12: Definição da função auxiliar de multiplicação



```

1 umulaux :: Magnitude -> Int -> Int -> Magnitude
2 umulaux [] y 0 = 0
3 umulaux [] y c = [c]
4 umulaux (x : xs) y c
5   | res > 9 = res `mod` 10 : umulaux xs y (res `div` 10)
6   | otherwise = res : umulaux xs y 0
7   where
8     res = x * y + c
9
10 umulterms :: Magnitude -> Magnitude -> [Magnitude]
11 umulterms x y = [reverse (umulaux (reverse x) (reverse y !! n) 0) ++ replicate n 0 | n <- [0 .. length y - 1]]
12
13 umul :: Magnitude -> Magnitude -> Magnitude
14 umul x y
15   | length x < length y = rmlleadzeros (foldl usoma [] (umulterms x y))
16   | otherwise = rmlleadzeros (foldl usoma [] (umulterms y x))

```

Para a multiplicação de magnitudes, é utilizada a lógica de multiplicação primária com campo de *carry*, tal como a soma, diferindo apenas no facto que os campos são multiplicados e que o campo *carry* corresponde à divisão inteira dos dígitos multiplicados em cada iteração. Colocando a menor magnitude no segundo campo, itera-se todos os dígitos deste segundo número por cada um dos dígitos do primeiro número, e para cada dígito do segundo número é gerado uma lista de dígitos. Estas listas, que representam resultados parciais da multiplicação, são depois devidamente somados, tendo em conta a ordem de magnitude que representam: a primeira lista tem ordem 0, pelo que não é adicionado nenhum zero à direita; a segunda tem ordem 1, pelo que é adicionado 1 zero à direita; e assim por diante.

Tomando um caso exemplo, para multiplicar 323 com 142, o processo é o seguinte:

1. Primeiro, é calculada a lista de dígitos da multiplicação de 323 com 2, que resulta em [6,4,6];
2. De seguida, é calculada a lista de dígitos da multiplicação de 323 com 4, que resulta em [1,2,9,2];
3. Por fim, é calculada a lista de dígitos da multiplicação de 323 com 1, que resulta em [3,2,3];
4. À primeira lista não é adicionado nenhum zero à direita, pois é de ordem de magnitude 0.
5. À segunda lista é adicionado 1 zero à direita, pois tem ordem de magnitude 1, do que resulta a lista [1,2,9,2,0].
6. À terceira e última lista são adicionados 2 zeros à direita, pois tem ordem de magnitude 2, do que resulta a lista [3,2,3,0,0].
7. Por fim, são somadas as listas [6,4,6], [1,2,9,2,0] e [3,2,3,0,0], resultando em [4,5,8,6,6].

1.2.7 Definição da função divBN

Figura 13: Definição da função divBN

```

1  divBN :: BigNumber -> BigNumber -> (BigNumber, BigNumber)
2  divBN x y
3    | not (sign x) || not (sign y) = error "divBN does not support negative arguments"
4    | otherwise = (True, quotient), (True, remainder))
5  where
6    quotient = udiv (mag x) (mag y)
7    remainder = usub (mag x) (umul quotient (mag y))

```

A função *divBN* calcula a divisão inteiro e respetivo resto de dois BigNumbers. Tal como especificado, não é aceite a divisão envolvendo números negativos, pelo que o sinal não entra em questão no cálculo do resultado. Para o cálculo do quociente, é utilizada uma função auxiliar (abaixo aprofundada). Para o cálculo do resto é utilizado o quociente, tomando partido da propriedade *resto = dividendo - (divisor x quociente)*.

Figura 14: Definição da função auxiliar de multiplicação

```

1  umulaux :: Magnitude -> Int -> Int -> Magnitude
2  umulaux [] y 0 = []
3  umulaux [] y c = [c]
4  umulaux (x : xs) y c
5    | res > 9 = res `mod` 10 : umulaux xs y (res `div` 10)
6    | otherwise = res : umulaux xs y 0
7  where
8    res = x * y + c
9
10 umulterms :: Magnitude -> Magnitude -> [Magnitude]
11 umulterms x y = [reverse (umulaux (reverse x) (reverse y !! n) 0) ++ replicate n 0 | n <- [0 .. length y - 1]]
12
13 umul :: Magnitude -> Magnitude -> Magnitude
14 umul x y
15   | length x < length y = rmlleadzeros (foldl usoma [] (umulterms x y))
16   | otherwise = rmlleadzeros (foldl usoma [] (umulterms y x))

```

Para a divisão de magnitudes, é utilizada a lógica de divisão primária com campo de *dividendo atual temporário*. Começa-se por iterar os dígitos do dividendo, do de maior ordem de magnitude para o de menor. Sempre que seja possível dividir o *dividendo atual temporário*, é calculado o número de vezes que o *divisor* divide esse campo, e é devolvido esse valor. O *dividendo atual temporário* passa a ser o resto dessa divisão parcial. Caso não seja possível, adiciona-se o dígito atual ao *dividendo atual temporário* e é devolvido 0. Este processo é efetuado até se consumir todos os dígitos do dividendo.

Tomando um caso exemplo, para dividir 3234 por 14, o processo é o seguinte:

1. Primeiro verifica-se se [] é menor que [1,4]. Como é, não é divisível e adiciona-se 3 ao *dividendo atual temporário*, devolvendo a soma de [0] com a chamada recursiva.
2. De seguida verifica-se se [3] é menor que [1,4]. Como é, adiciona-se 2 ao *dividendo atual temporário*, devolvendo a soma de [0,0] com a chamada recursiva.
3. Verifica-se se [3,2] é menor que [1,4]. Como não é, calcula-se a divisão de 32 por 14, resultando quociente 2 e resto 4. Assim, o *dividendo atual temporário* passa a ser [4,3], e devolve-se a soma de [0,0,2] com a chamada recursiva.
4. Verifica-se se [4,3] é menor que [1,4]. Como não é, calcula-se a divisão de 43 por 14, resultando quociente 3 e resto 1. Assim, o *dividendo atual temporário* passa a ser [1,4], e devolve-se a soma de [0,0,2,3] com a chamada recursiva.
5. Por fim, verifica-se se [1,4] é menor que [1,4]. Como não é, faz-se a divisão, resultando em quociente 1 e resto 0. Como se chegou ao fim do dividendo, é devolvido [0,0,0,2,3,1] - os *leading zeros* são tratados por outra função.

1.3 Extensão do Módulo de Fibonacci a BigNumber

1.3.1 Definição da função fibRecBN

Figura 15: Definição da função fibRecBN

```

1 fibRecBN :: BigNumber -> BigNumber
2 fibRecBN (False, _) = error "Negative BigNumber on fibRecBN!"
3 fibRecBN (True, [0]) = scanner "0"
4 fibRecBN (True, [1]) = scanner "1"
5 fibRecBN n = fibRecBN (n `subBN` scanner "1") `somaBN` fibRecBN (n `subBN` scanner "2")

```

A adaptação da função a BigNumber é trivial. Apenas foi necessário substituir as operações aritméticas pelas do módulo BigNumber, e o funcionamento e output da função é idêntico.

1.3.2 Definição da função fibListaBN

Figura 16: Definição da função fibListaBN

```

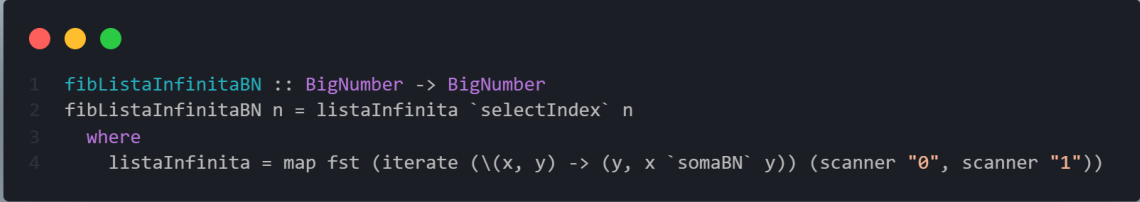
1 genInteirosFrom2 :: BigNumber -> [BigNumber]
2 genInteirosFrom2 x = init (take (convToInt x) (iterate (\x -> x `somaBN` scanner "1") (scanner "2")))
3
4 fibListaBN :: BigNumber -> BigNumber
5 fibListaBN n =
6   let lista =
7     scanner "0" :
8     scanner "1" :
9     [ (lista `selectIndex` (x `subBN` scanner "1"))
10      `somaBN` (lista `selectIndex` (x `subBN` scanner "2"))
11      | x <- genInteirosFrom2 n
12    ]
13   in last lista

```

A adaptação da função a BigNumber foi, no geral, também trivial. Além de alterar as operações aritméticas para as do módulo BigNumber, foi necessário criar funções de seleção de índice (*selectIndex*) e de conversão para inteiro (*convToInt*, esta apenas usada para utilizar funções do prelúdio padrão). Além disso, foi necessário gerar uma lista de BigNumbers, para a compreensão em lista da função *fibListaBN*, utilizando a função *iterate*.

1.3.3 Definição da função fibListaInfinitaBN

Figura 17: Definição da função fibListaInfinitaBN

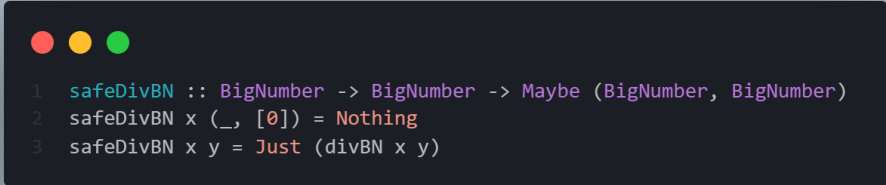


```
1 fibListaInfinitaBN :: BigNumber -> BigNumber
2 fibListaInfinitaBN n = listaInfinita `selectIndex` n
3   where
4     listaInfinita = map fst (iterate (\(x, y) -> (y, x `somaBN` y)) (scanner "0", scanner "1"))
```

A adaptação da função a `BigNumber` é, tal como *fibRecBN*, trivial, apenas alterando as operações aritméticas para as do módulo `BigNumber` - e usando a função *selectIndex* previamente referida.

1.4 Função de Divisão Segura

Figura 18: Definição da função safeDivBN



```
1 safeDivBN :: BigNumber -> BigNumber -> Maybe (BigNumber, BigNumber)
2 safeDivBN x (_, [0]) = Nothing
3 safeDivBN x y = Just (divBN x y)
```

Tal como pedido, é criado o padrão para proteger da divisão por 0, tomando partido do monad *Maybe*. Em todos os outros casos, o resultado é do da divisão normal não protegida (*divBN*).

2 Casos de Teste

Abaixo, apresentam-se vários casos de teste para todas as funções supracitadas.

2.1 fibRec

Chamada da Função	Resultado	Tempo (ms)
fibRec 0	0	32
fibRec 1	1	32
fibRec 8	21	33
fibRec 30	832040	289
fibRec 40	102334155	28607

2.2 fibLista

Chamada da Função	Resultado	Tempo (ms)
fibLista 0	0	31
fibLista 1	1	31
fibLista 8	21	32
fibLista 30	832040	33
fibLista 150	9969216677189303386214405760200	33
fibLista 1000	434665576869374564...66849228875	39
fibLista 10000	33644764876431783...59947366875	502
fibLista 100000	2597406934722172...53428746875	157016

2.3 fibListaInfinita

Chamada da Função	Resultado	Tempo (ms)
fibListaInfinita 0	0	31
fibListaInfinita 1	1	31
fibListaInfinita 8	21	31
fibListaInfinita 30	832040	32
fibListaInfinita 150	9969216677189303386214405760200	32
fibListaInfinita 1000	434665576869374564...66849228875	32
fibListaInfinita 10000	33644764876431783...59947366875	34
fibListaInfinita 100000	2597406934722172...53428746875	220

2.4 scanner

Chamada da Função	Resultado
scanner "0"	(True, [0])
scanner "1"	(True, [1])
scanner "-1"	(False, [1])
scanner "--"	Invalid BigNumber from string '--'
scanner "--"	Invalid BigNumber from string '--'
scanner "-112312"	(False, [1,1,2,3,1,2])
scanner "112312"	(True, [1,1,2,3,1,2])
scanner "112a312"	Invalid BigNumber from string '112a312'

2.5 output

Chamada da Função	Resultado
output (True, [0])	"0"
output (True, [1])	"1"
output (False, [1])	"-1"
output (False, [])	Error
output (False, [1,1,2,3,1,2])	"-112312"
output (True, [1,1,2,3,1,2])	"112312"
output (True, [1,1,2,a,3,1,2])	Error

2.6 somaBN

Chamada da Função	Resultado
somaBN (scanner "0") (scanner "0")	(True, [0])
somaBN (scanner "1") (scanner "-1")	(True, [0])
somaBN (scanner "1") (scanner "1")	(True, [2])
somaBN (scanner "99") (scanner "1")	(True, [1,0,0])
somaBN (scanner "99") (scanner "-100")	(False, [1])
somaBN (scanner "55") (scanner "-32")	(True, [2,3])
somaBN (scanner "-41") (scanner "-99")	(False, [1,4,0])

2.7 subBN

Chamada da Função	Resultado
subBN (scanner "0") (scanner "0")	(True, [0])
subBN (scanner "1") (scanner "-1")	(True, [2])
subBN (scanner "1") (scanner "1")	(True, [0])
subBN (scanner "99") (scanner "1")	(True, [9,8])
subBN (scanner "99") (scanner "-100")	(True, [1,9,9])
subBN (scanner "-55") (scanner "-32")	(False, [2,3])
subBN (scanner "-41") (scanner "-99")	(True, [5,8])

2.8 mulBN

Chamada da Função	Resultado
mulBN (scanner "0") (scanner "0")	(True, [0])
mulBN (scanner "1") (scanner "-1")	(False, [1])
mulBN (scanner "1") (scanner "1")	(True, [1])
mulBN (scanner "99") (scanner "1")	(True, [9,9])
mulBN (scanner "99") (scanner "-100")	(False, [9,9,0,0])
mulBN (scanner "-55") (scanner "-32")	(True, [1,7,6,0])
mulBN (scanner "-41") (scanner "99")	(False, [4,0,5,9])

2.9 divBN

Chamada da Função	Resultado
divBN (scanner "1") (scanner "0")	((True, [0]), (True, [1]))
divBN (scanner "1") (scanner "-1")	divBN does not support negative arguments
divBN (scanner "99") (scanner "1")	((True, [9,9]), (True, [0]))
divBN (scanner "99") (scanner "2")	((True, [4,9]), (True, [1]))
divBN (scanner "99") (scanner "100")	((True, [0]), (True, [9,9]))
divBN (scanner "55") (scanner "32")	((True, [1]), (True, [2,3]))
divBN (scanner "4211") (scanner "99")	((True, [4,2]), (True, [6,3]))

2.10 fibRecBN

Chamada da Função	Resultado	Tempo (ms)
fibRecBN (scanner "0")	(True, [0])	32
fibRecBN (scanner "1")	(True, [1])	32
fibRecBN (scanner "8")	(True, [2,1])	32
fibRecBN (scanner "30")	(True, [8,3,2,0,4,0])	6145
fibRecBN (scanner "35")	(True, [9,2,2,7,4,6,5])	66070

2.11 fibListaBN

Chamada da Função	Resultado	Tempo (ms)
fibListaBN (scanner "0")	0	32
fibListaBN (scanner "1")	1	32
fibListaBN (scanner "8")	21	32
fibListaBN (scanner "30")	832040	32
fibListaBN (scanner "150")	9969216677189303386214405760200	35
fibListaBN (scanner "1000")	434665576869374564...66849228875	67
fibListaBN (scanner "10000")	33644764876431783...59947366875	3665

2.12 fibListaInfinitaBN

Chamada da Função	Resultado	Tempo (ms)
fibListaInfinitaBN 0	0	35
fibListaInfinitaBN 1	1	33
fibListaInfinitaBN 8	21	33
fibListaInfinitaBN 30	832040	35
fibListaInfinitaBN 150	9969216677189303386214405760200	32
fibListaInfinitaBN 1000	434665576869374564...66849228875	52
fibListaInfinitaBN 10000	33644764876431783...59947366875	2058

2.13 safeDivBN

Chamada da Função	Resultado
divBN (scanner "0") (scanner "0")	Nothing
divBN (scanner "11") (scanner "0")	Nothing
divBN (scanner "11") (scanner "1")	Just ((True, [1,1]), (True, [0]))
divBN (scanner "11231") (scanner "21")	Just ((True, [5,3,4]), (True, [1,7]))

3 Alínea 4 - Análise e Comparação das resoluções das alíneas 1 e 3

É importante definir, desde logo, os limites de cada uma das representações.

Int, em Haskell, pertence à **type-class** *Bounded*, o que significa que é um tipo com limite inferior e superior. Segundo a documentação, um *Int* é 'Um tipo de inteiro de precisão fixa com representação mínima de $[-2^{29}, 2^{29} - 1]$ '.

Já *Integer*, ao contrário de *Int*, não pertence à **type-class** *Bounded*, e é um tipo de precisão arbitrária: pode representar um número tão grande quanto a memória o permitir.

O mesmo se aplica ao tipo criado *BigNumber*, que apenas tem como limite de representação a memória para armazenar a lista de dígitos.

Com isto em mente, ficam claros os limites de representação de cada uma dos tipos.

Nas funções em questão, nota-se, desde logo, a limitação quando aplicada ao tipo *Int*: já é incapaz de representar *fibLista 93*, por exemplo. No entanto, tanto *Integer* como *BigNumber* não parecem ter problemas quanto ao limite/grandeza do número que representam.

Fica claro, assim, que para a representação de números grandes, como é o caso da sequência de Fibonacci para números superiores a 100, p.e., *Integer* ou *BigNumber* são melhores escolhas do que *Int*, dadas as suas restrições.