Trabalho Prático 1

Grupo: G2_01

Realizado por:

- Diogo Miguel Chaves dos Santos Antunes Pereira – up201906422
- Joana Teixeira Mesquita - up201907878

BigNumbers.hs

- data BigNumber
  - O BigNumber (BigNumber {bn :: [Int], neg:: bool}) é um datatype para a representação de números positivos e negativos:
    - representa o valor absoluto do número numa lista de números inteiros (chamada de bn) em que cada algarismo está separado por uma vírgula;
    - representa o sinal do número através de um booleano (que se chama neg) que, se tiver o valor “true” indica que o número é negativo e se tiver o valor “false” indica que o número é positivo.

- scanner
  - A função recebe uma “string” que representa um número (para números negativos a “string” tem de ter como primeiro caracter ‘-’) e transforma-a num BigNumber removendo zeros extra à esquerda, se existirem.
  - Se o formato da string for inválido ocorre uma exceção.
  - Casos de teste:
    - scanner “50”:
      - output: 50
      - gera o BigNumber {bn: [5,0], neg = false}
    - scanner “-50”:
      - output: -50
      - gera o BigNumber {bn: [5,0], neg = true}
    - scanner “00030”:
      - output: 30
      - gera o BigNumber {bn: [5,0], neg = false}
    - scanner “-5999999999939392933332222”:
      - output: 5999999999939392933332222
      - gera o BigNumber {bn: [5,9,9,9,9,9,9,9,9,9,9,3,9,3,9,2,9,3,3,3,3,2,2,2,2], neg = true}
    - scanner “59999.999999393929”:

        - *** Exception: Prelude.read no parse
    - scanner “”:
        - *** Exception: Prelude.read no parse

- output
    - A função recebe um BigNumber e transforma-o numa string removendo zeros extra à esquerda, se existirem.
    - Esta função também foi colocada como a função default quando um BigNumber é retornado por uma função.
    - Casos de teste:
        - output(BigNumber {bn = [0], neg = False}) = “0”
        - output(BigNumber {bn = [1, 0], neg = True}) = “-10”
        - output(BigNumber {bn = [9, 0, 2, 5, 3, 8, 8, 7, 6, 9, 0, 6, 2, 4], neg = False}) = "90253887690624"

- somaBN
    - Esta função recebe dois BigNumbers:
        - caso tenham o mesmo sinal, soma os módulos e atribui ao resultado este sinal;
        - caso tenham sinais diferentes chama a função de subtração subBN com o BigNumber positivo como primeiro argumento e o módulo do BigNumber negativo como segundo argumento.
    - A função somaBN começa por chamar a função sumCarry com os bn dos argumentos. Dentro desta função é chamada a função sumLists que inverte e soma os BigNumbers com as funções do prelúdio reverse e zipWith. Se as listas tiverem tamanhos diferentes, para não perder algarismos devido à função zipWith, são adicionados zeros à lista menor até as duas terem o mesmo tamanho. Com o resultado desta função:
        - são colocados numa lista os carries (excessos gerados pela soma) com a função calculateCarries a partir da divisão inteira com 10;
        - é colocado numa lista o resultado com os carries removidos com o auxílio da função removeCarries a partir do módulo com 10.
    - Estas duas listas serão somadas pela função addCarries, que coloca um 0 à direita do resultado do removeCarries caso a ordem do número aumente e um 0 à esquerda do resultado do calculateCarries para que os elementos deste fiquem alinhados com os números a que vão ser somados.
    - Por fim, como a adição dos carries pode levar a novos carries a função sumCarry é recursiva e só parará quando o calculateCarries retornar uma lista de zeros.
    - Ao retornar à função sumBN inverte o resultado recebido (já que o número foi invertido no sumLists) e remove zeros extras no início do número.
    - Casos de teste:
        - somaBN (scanner “999”) (scanner “11”) = 1010

        - somaBN (scanner “-99”) (scanner “-11”) = -110
        - somaBN (scanner “999999”) (scanner “11”) = 1000010
        - somaBN (scanner “999999”) (scanner “-11”) = 999988 (chama a função subBN)
        - somaBN (scanner “0”) (scanner “24”) = 24

- subBN
    - Esta função recebe dois BigNumbers:
        - caso tenham o mesmo sinal, subtrai os BigNumbers em módulo e atribui ao resultado o sinal do argumento com maior módulo (tendo em atenção que a operação muda o sinal do segundo argumento Ex.: 1 - (+2) = 1 - 2, -1 - (-2) = -1 + 2);
        - caso tenham sinais diferentes chama a função de soma somaBN com o sinal do segundo argumento trocado.
    - A função subBN começa por chamar a função subCarry com os bn dos argumentos. Dentro desta função é chamada a função subLists que inverte e subtrai os BigNumbers com as funções do prelúdio reverse e zipWith. Se as listas tiverem tamanhos diferentes, para não perder algarismos devido à função zipWith, são adicionados zeros à lista menor até as duas terem o mesmo tamanho. Com o resultado desta função:
        - são colocados numa lista os carries (faltas geradas pela subtração devido ao facto do algarismo do primeiro número ser menor que o algarismo de mesma ordem do segundo) com a função calculateCarriesSub que coloca 1 na mesma posição que os números negativos da lista criada pelo subList e 0 nos restantes;
        - é colocado numa lista o resultado com os carries removidos com o auxílio da função removeCarries a partir do módulo com 10.
    - Estas duas listas serão subtraídas pela função subtractCarries, que coloca um 0 à direita do resultado do removeCarries caso a ordem do número aumente e um 0 à esquerda do resultado do calculateCarriesSub para que os elementos destes fiquem alinhados com os números a que vão ser subtraídos.
    - Por fim, como a subtração dos carries pode levar a novos carries, a função subCarry é recursiva e só parará quando o calculateCarries retornar uma lista de zeros.
    - Ao retornar à função subBN inverte o resultado recebido (já que o número foi invertido no subLists) e remove zeros extras no inicio do número.
    - Casos de teste:
        - subBN (scanner “999”) (scanner “11”) = 988
        - subBN (scanner “-99”) (scanner “-11”) = -88
        - subBN (scanner “999999”) (scanner “11”) = 999988
        - subBN (scanner “999999”) (scanner “-11”) = 1000010 (chama a função somaBN)
        - subBN (scanner “0”) (scanner “24”) = -24

- mulBN
  - Esta função recebe dois BigNumbers:
    - caso tenham o mesmo sinal atribui ao resultado sinal positivo;
    - caso tenham sinais diferentes atribui ao resultado sinal negativo.
  - A função começa por multiplicar cada dígito do primeiro argumento com todos os elementos do segundo argumento, utilizando a função mulLists que retorna a lista, na qual eles se encontram, por ordem do seu cálculo.
  - Esta lista é de seguida usada para a criação de uma lista de listas que agrupa os valores em listas com o tamanho do segundo argumento. Isto é feito com a função splitEvery que divide uma lista de n em n argumentos.
  - Após esta separação estar feita é chamada a função sumOfMul que adiciona o número de zeros igual ao index da lista na lista de listas, para estas serem somadas corretamente e depois soma-as, recursivamente, com somaBN.
  - Quando o resultado retorna à função mulBN que antes de terminar, remove zeros extra à esquerda do número.
  - Casos de teste:
    - mulBN (scanner “999”) (scanner “11”) = 10989
    - mulBN (scanner “-99”) (scanner “-11”) = 10989
    - mulBN (scanner “-27”) (scanner “180”) = -4860
    - mulBN (scanner “999999”) (scanner “11”) = 10999989
    - mulBN (scanner “0”) (scanner “24”) = 0
    - mulBN (scanner "999999") (scanner "92233720368547758071") = 92233628134827389523241929

- divBN
  - Esta função recebe dois BigNumbers:
    - caso tenham o mesmo sinal atribui ao resultado sinal positivo;
    - caso tenham sinais diferentes atribui ao resultado sinal negativo;
    - nota: Se os BigNumbers forem iguais retorna imediatamente (1,0) ou (-1,0) conforme os sinais, se o dividendo for menor que o divisor retorna imediatamente 0 como quociente e calcula o resto a partir dos dois números.
  - Primeiro é chamada a função assembleDivision com o bn de cada BigNumber que calcula recursivamente o quociente da divisão:
    - começa por chamar a função getLowestFit para descobrir qual é a menor porção do dividendo que pode ser dividida pelo divisor, criando uma lista de listas em que o primeiro elemento é essa porção do número e o segundo é o resto do número;
    - de seguida descobre através da função calcQuo, que realiza uma pesquisa número a número, qual o quociente da divisão entre a porção do número divisível descoberta anteriormente e o nosso divisor;

        - seguidamente é multiplicado este quociente pelo divisor e é subtraído à porção do número divisível;
        - este resultado é de seguida colocado novamente numa lista de listas em que o segundo argumento é mais uma vez o resto do número que ainda não foi dividido e o primeiro é o resultado da subtração anterior;
        - estas duas listas novamente colocados numa única lista a partir da função quotJoinRemainder, que também remove os zeros extra à esquerda resultantes da subtração;
        - se o resultado da diferença com o algarismo seguinte do dividendo for divisível pelo divisor é retornado o quociente e é chamada novamente a função para o que resta do dividendo;
        - se pelo contrário isto não acontecer é retornado, juntamente com o quociente e a chamada da função, um zero à esquerda do quociente de acordo com as regras da divisão;
        - as chamadas recursivas desta função são concatenadas de modo que o produto final tenha o quociente completo da divisão;
        - a recursão termina quando o número que se encontra no denominador é menor que o divisor. Retorna -1 para indicar o fim do quociente e o denominador que representa o resto da divisão.
    - Quando a divBN recebe este resultado, coloca em primeiro lugar num tuple a porção deste que representa o quociente e em segundo lugar a porção que representa o resto retornando esse tuple.
    - Casos de teste:
        - divBN (scanner “999”) (scanner “11”) = (90,9)
        - divBN (scanner “-99”) (scanner “11”) = (-9,0)
        - divBN (scanner “-180”) (scanner “-27”) = (6,18)
        - divBN (scanner “12”) (scanner “24”) = (0,12)
        - divBN (scanner "2") (scanner "2") = (1,0)

- safeDivBN
    - Esta função recebe dois BigNumbers:
        - caso o divisor seja 0 a função retorna Nothing;
        - caso contrário é chamada a função divBN normalmente.
    - Casos de teste:
        - safeDivBN (scanner “999”) (scanner “11”) = (90,9)
        - safeDivBN (scanner “150”) (scanner “0”) = Nothing

Fib.hs

- fibRec
  - A função calcula os números de fibonacci recursivamente.
  - Para cada n, chama-se a si própria para calcular os números de fibonacci (n-1) e (n-2) que somados darão o número de fibonacci n.
  - Casos de teste:
    - fibRec 2 = 1
    - fibRec 10 = 55
    - fibRec 30 = 83204
    - Para números mais elevados a função apresenta desempenho temporal muito fraco.

- fibLista
  - A função calcula os números de fibonacci com *memoization*.
  - Sempre que um novo n é calculado é inserido numa lista de forma a evitar cálculos futuros repetidos.
  - Para a função ser mais rápida o n na indexação é omitido pois se for chamado explicitamente tem sempre de ser calculado mesmo quando não é necessário.
  - Casos de teste:
    - fibLista 2 = 1
    - fibLista 10 = 55
    - fibLista 30 = 83204
    - fibLista 100 = 354224848179261915075
    - fibLista 200 = 280571172992510140037611932413038677189525
    - A função fibLista retorna o resultado quase instantaneamente até n ordem de 10^5

- fibListaInfinita
  - A função calcula todos os números de fibonacci colocando-os numa lista infinita e seguidamente retorna o número no index n da lista.
  - O cálculo desta lista recorre a programação dinâmica em que, para calcular cada número de fibonacci a função acede aos dois elementos anteriores na lista somando-os e guardando o valor na posição de index atual.
  - Casos de teste:
    - fibListaInfinita 2 = 1
    - fibListaInfinita 10 = 55
    - fibListaInfinita 30 = 83204
    - fibListaInfinita 100 = 354224848179261915075
    - fibListaInfinita 200 = 280571172992510140037611932413038677189525

        - A função fibListaInfinita retorna o resultado quase instantaneamente até n ordem de 10^6

- fibRecBN
    - A função calcula os números de fibonacci no formato BigNumber recursivamente, sendo logicamente igual ao fibRec.
    - Casos de teste:
        - fibRecBN (scanner ”2”) = 1
        - fibRecBN (scanner “10”) = 55
        - fibRecBN (scanner “20”) = 6765
        - Para BigNumbers mais elevados a função apresenta desempenho temporal muito fraco.

- fibListaBN
    - A função calcula os números de fibonacci no formato BigNumber com *memoization*, comportando-se de maneira semelhante à função fibLista.
    - A única diferença lógica entre as duas funções é o facto de que, um BigNumber não pode ser usado para indicar um index numa lista, por isso a indexação é explícita, afetando o desempenho da função.
    - Casos de teste:
        - fibListaBN (scanner ”2”) = 1
        - fibListaBN (scanner “10”) = 55
        - fibListaBN (scanner “20”) = 6765
        - Para BigNumbers mais elevados a função apresenta desempenho temporal muito fraco.

- fibListaInfinitaBN
    - A função calcula todos os números de fibonacci no formato BigNumber colocando-os numa lista infinita e seguidamente retorna o número no index n da lista. O funcionamento desta função é muito semelhante ao da função fibListaInfinita.
    - Casos de teste:
        - fibListaInfinitaBN (scanner ”2”) = 1
        - fibListaInfinitaBN (scanner ”10”) = 55
        - fibListaInfinitaBN (scanner ”20”) = 6765
        - fibListaInfinitaBN (scanner ”100”) = 354224848179261915075
        - A função fibListaInfinita retorna o resultado quase instantaneamente até n ordem de 10^3

Pergunta 4

Nas funções de fibonacci de Int para Int, o número máximo é limitado pelo tamanho máximo de um Int: maxBound = 9223372036854775807, logo teoricamente o maior número que estas funções podem ter como argumento é fib 92 = 7540113804746346429.

Nas funções de fibonacci de Integer para Integer tal como nas que utilizam BigNumbers, o número máximo é limitado pela memória disponível, que em haskell é tipicamente 2^(2^37), logo o maior número que estas funções podem ter como argumento é o Integer que leve ao maior resultado possível que seja menor do que 2^(2^37).