Trabalho Prático 1 de PFL

Representação:

Para este trabalho, decidimos criar 3 novos types em haskell:

- **Var**, um tuplo de 2 elementos: um caracter representativo de uma incógnita e um número Int representativo do seu respetivo expoente, ou seja **type Var = (Char, Int)**.
  Por exemplo, um Var de (‘x’, 3) representa x^3

- **Vars**, uma lista de Var, representativa das incógnitas (e seus expoentes) presentes no monómio, ou seja **type Vars = [Var]**
  Por exemplo, um Vars de [(‘x’, 1), (‘y’,4), (‘z’, 3)] representa x(y^4)(z^3)

- **Mon**, representativo de um monómio, um tuplo de 2 elementos: um número Int representativo do coeficiente e um Vars, ou seja **type Mon = (Int, Vars)**. Deste modo, o monómio pode conter qualquer quantidade de incógnitas, visto que estão guardadas na lista Vars de tamanho variável.
  Por exemplo, um Mon de (2, [(‘x’, 1), (‘y’,4), (‘z’, 3)]) representa o monómio 2x(y^4)(z^3)

- **Pol**, representativo de um polinómio, pode ser descrito simplesmente como uma lista de Mon (monómios), ou seja **type Pol = [Mon]**.
  Por exemplo, um Pol de [(3, [(‘x’, 2)]), (-5, [(‘y’, 3), (‘z’, 1)])] representa o polinómio 3(x^2) - 5(y^3)z

Implementação:

Normalização

A normalização de um polinómio foi implementada através da chamada de funções recursivas que realizam pequenas tarefas. Estas funções são executadas pela seguinte ordem:

- **fixExpZero** que determina se existem monómios com alguma incógnita que esteja elevada a 0, e se sim, remove essas incógnitas. Esta função chama funções de auxílio:
  - substituteExpZero (realiza o necessário para quando a incógnita de expoente 0 é a única do monómio)
  - removeExpZero (realiza o necessário para quando a incógnita de expoente 0 não é a única do monómio)
  - o que ajudam a segmentar as tarefas
- **sortVars** que ordena alfabeticamente as incógnitas dentro de cada monómio do polinómio
- **sortPol** que organiza os monómios conforme a ordem alfabética das suas incógnitas (como parâmetro secundário são usados os expoentes).
- **addMons** que determina se o polinómio contém monómios possíveis de somar, e se sim, soma-os. Esta função chama funções de auxílio:

    - findX (descobre se o polinómio tem algum monómio x que possa ser somado ao monómio n)
    - addSome (soma os coeficientes de 2 monómios e retorna 1 só)
- **removeZero** que remove do polinómio todos os monómios que tenham coeficiente 0.

Assim, a normalização de um polinómio pode ser corretamente executada. Para a utilizar, basta usar chamar a função **normalize** e indicar um polinómio, ou seja **normalize pol1**
Por exemplo: para normalizar 4y - 2(y^2) + 3(x^2) + yx + x^2 seria preciso introduzir normalize [(4, [('y', 1)]), (-2, [('y', 2)]), (3, [('x', 2)]), (1, [('y', 1), ('x', 1]), (1, [('x', 2)])]

Adição

Visto que a normalização já realiza a adição dos monómios, tudo o que aqui é necessário fazer para somar dois polinómios é concatená-los (visto que são listas de monómios) e realizar a normalização dessa concatenação.

Assim, a adição de dois polinómios pode ser corretamente executada. Para a utilizar, basta usar a função **addPols** e indicar dois polinómios, ou seja **addPols pol1 pol2**
Por exemplo: para somar o polinómio 2x + 4y + 3(z^2) com o polinómio -3x + z + y^2 + z^2 seria preciso introduzir addPols [(2, [('x', 1)]), (4, [('y', 1)]), (3, [('z', 2)])] [(-3, [('x', 1)]), (1, [('z', 1)]), (1, [('y', 2)]), (1, [('z', 2)])]

Multiplicação

Para a multiplicação de dois polinómios utilizamos duas funções principais:

- **multi**, a própria função de multiplicação, que coordena as multiplicações entre cada monómio de um polinómio com os monómios do outro. Esta função possui funções de auxílio:
    - multMons (multiplica dois monómios)
    - multVars (multiplica Vars compatíveis de um monómio)
    - addExp (soma expoentes de 2 Vars)
    - findVar (descobre se o monómio tem algum Var x que possa ser somado ao Var n)
- **normalize**, que garante que o polinómio obtido pela multiplicação está normalizado.

Assim, a multiplicação de dois polinómios pode ser corretamente executada. Para a utilizar, basta usar a função **multPols** e indicar dois polinómios, ou seja **multPols pol1 pol2**
Por exemplo: para multiplicar o polinómio 2x + 4y + 3(z^2) com o polinómio -3x + z + y^2 + z^2 seria preciso introduzir multPols [(2, [('x', 1)]), (4, [('y', 1)]), (3, [('z', 2)])] [(-3, [('x', 1)]), (1, [('z', 1)]), (1, [('y', 2)]), (1, [('z', 2)])]

Derivação

Para a derivação de um polinómio utilizamos:

- **compreensão de listas** para essencialmente chamar para cada monómio a função **derivateMon** que o irá derivar por uma incógnita. Esta função chama funções de auxílio:
  - findN (determina se existe na lista existe um var com incógnita n)
  - removeN (retorna a lista de vars sem o var de incógnita n)
  - getExpN (retorna o expoente do var da lista que tem a incógnita n, sendo que já sabemos que ele existe algures na lista)
- **normalize**, que garante que o polinómio obtido pela sua derivação está normalizado.

Assim, a derivação de um polinómio pode ser corretamente executada. Para a utilizar, basta usar a função **derivatePol** e indicar um Char representativo da incógnita pela qual se vai derivar e um polinómio, ou seja **derivatePol char1 pol1**
Por exemplo: para derivar 4y - 2(y^2) + 3(x^2) + yx + x^2 por x seria preciso introduzir derivatePol ‘x’ [(4, [(‘y’, 1)]), (-2, [(‘y’, 2)]), (3, [(‘x’, 2)]), (1, [(‘y’, 1), (‘x’, 1]), (1, [(‘x’, 2)])]

Parse

Para o parse de um polinómio utilizamos:

- **removeWhiteSpace** responsável por remover os espaços existentes na string
- **readPolynomial** responsável por passar um String para e o nosso type Pol onde separa o polinômio a partir dos sinais + ou - e chama funções de auxilio sobre cada um dos componentes
  - readMonomial (retorna o monômio separa o número mais sinal do char)
  - parseCoef (retorna o coeficiente mais o sinal para ser usado na função readMonomial)
  - checkEmptyList (verifica se não existem variáveis e coloca um número isolado na forma (‘ ’,1) e chama a função readVarList que trata das variáveis)
- **parsePol**, percorre a lista do Pol chamando depois a função parsePol’ que irá tratar da recursividade e dos sinais entre os monômios
  - parseMon (retorna o monômio em forma de string este acrescenta os ‘^’ necessários a monômios que só tem uma variável ao que são maiores ele chama a função parseVar que trata de pôr as variáveis a multiplicar umas pelas outras.)

Test

Este módulo é responsável por testar o nosso código e todas as suas funcionalidades contendo também uma gui para facilitar o input de polinômios e todas as operações de forma a ser mais intuitivo correr o código testar, e usufruir das funcionalidades por nós implementadas.