Trabalho 1 - PFL

Este trabalho tem como objetivo desenvovler um programa Haskell que permita a manipulação simbólica de polinómios. Mais concretamente, devem ser implementadas as operações de soma, multiplicação e de derivação.

Para uma utilização mais fluida do programa, a conversão de uma string de input para o formato de representação interno e a conversão deste de volta para uma string de output devem ser implementadas.

Funcionalidades Implementadas

Todas as funcionalidades previstas foram implementadas.

- [x] Parsing String → Polinómio
- [x] Normalização de polinómios
- [x] Soma de polinómios
- [x] Multiplicação de polinómios
- [x] Derivação de polinómios
- [x] Parsing Polinómio → String

Configuração / Instalação / Funcionamento

O código desenvolvido foi testando tanto em Linux (Ubuntu) como em Windows 11, tendo como ficheiro principal **Proj.hs**. O programa pode ser compilado e executado usando:

```
ghc *.hs
```

ou através do interpretador GHCI com:

```
ghci Proj.hs
main
```

Foi também criado um simples Makefile para facilitar a vida com as seguintes opções:

- `make` - compila o código e limpa os ficheiros temporários no final
- `make clean` - remove ficheiros temporários no final
- `make run` - correr ficheiro executável
- `make check` - correr testes ao código (mais sobre testagem abaixo)

Software Extra

Relativamente ao software, para além de todas as funcionalidades do Prelúdio, utilizamos também uma biblioteca para "Property Testing" do código chamada **QuickCheck**. Este pacote requer a instalação seguinte:

```
cabal install QuickCheck
```

Nota : Admite-se que o *cabal* já se encontre instalado na máquina.

Troubleshooting

Durante o desenvolvimento deste trabalho verificaram-se vários problemas ao instalar e utilizar o *QuickCheck* tanto em Windows como em Linux pelo que é possível que o código não corra.
Note-se, por exemplo, que o Makefile possui duas versões diferentes para compilar o código, dependendo do sistema operativo em uso.
Caso problemas se verifiquem, recomenda-se os seguintes passos:

```
import Prop_tests -- Comentar o import dos módulo de testes (módulo em Prop_tests.hs) presente em Proj.hs

-- Comentar igualmente as linhas em Proj.hs que chamam a testagem do código
main_test :: IO Bool
main_test = check
```

Por fim, pode ser removido o ficheiro "Prop_tests.hs" no qual se encontram os testes para evitar erros ao tentar fazer import do QuickCheck.
Deve agora ser possível compilar os ficheiros.

Testagem

Para testar o código recorremos, como já foi previamente mencionado, à ferramenta QuickCheck.
Por limitação de tempo e por não ser o foco principal do trabalho, fizemos apenas alguns testes para corroborar a solidez das operações principais.
Para correr os testes usamos o comando *make check*.

```
m:~/Desktop/1S/PFL/git_t1$ make check
echo "check" |  ghci Proj.hs
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
[1 of 4] Compiling Arithmetics      ( Arithmetics.hs, interpreted )
[2 of 4] Compiling Parser           ( Parser.hs, interpreted )
[3 of 4] Compiling Prop_tests       ( Prop_tests.hs, interpreted )
[4 of 4] Compiling Main             ( Proj.hs, interpreted )
Ok, four modules loaded.
*Main> === prop_associativity_sum from ./Prop_tests.hs:17 ===
+++ OK, passed 100 tests.

=== prop_null_element_sum from ./Prop_tests.hs:20 ===
+++ OK, passed 100 tests.

=== prop_coef_mult from ./Prop_tests.hs:25 ===
+++ OK, passed 100 tests.

=== prop_associativity_mult from ./Prop_tests.hs:28 ===
+++ OK, passed 100 tests.

=== prop_null_element_mult from ./Prop_tests.hs:31 ===
+++ OK, passed 100 tests.

=== prop_const_deriv from ./Prop_tests.hs:36 ===
+++ OK, passed 100 tests.

True
*Main> Leaving GHCi.
```

Representação Interna

Para representar polinómios e monómios, foram criados os seguintes tipos:

```
-- Types definition
type Monomio = ((Int, [Int]), String)
type Polinomio = [Monomio]
```

A sua representação é bastante intuitiva. Um polinómio nada mais é do que um conjunto de monómios que por sua vez é representado por algo deste género:

```
3x^2y = ((3,[2,1]), "xy") -- 3  x^2   y^1
```

Um monómio é constituído por um par entre um par de (Int,[Int]) com String. Cada char da string indica uma variável do monómio, estando o expoente de cada uma destas variáveis guardado em [Int] pela mesma ordem. Por fim, o primeiro Int representa o coeficiente.
Esta abordagem é simples e como é uma composição de pares, permite tomar partido das funções `fst` e `snd` para aceder rapidamente a cada elemento.

Estratégias de Implementação de Funcionalidades

Parsing Input

Para fazer um parser correto é necessário saber fazer a correta distinção entre operações e valores (polinómios). Add, Mult, Derive, Sub e Pow são as nossas possíveis operações e Poli o valor terminal para a conclução de uma intrepretação da expressão. Para nos facilitar a vida consideramos que o expoente do Pow, que serve como um expoente para polinomios e não para monómios, e a variável que é passada para o derive são ambas expressões que resultam num inteiro e num char representados por um polinómio.

```
data Expr = Add Expr Expr
          | Mult Expr Expr
          | Poli Polinomio
          | Derive Expr Expr
          | Sub Expr Expr
          | Pow Expr Expr
          deriving (Eq)

instance Show Expr where
  show x = poliParseToStr . normPoli . eval $ x
```

A função `eval` é então responsável por pegar na expressão e a reduzir ao valor terminal. No início surgiu a dúvida se uma linguagem ambígua não poderia dar asas a erros, porém a àrvore de sintaxe não será aqui construída nem teria, neste caso, impacto na evaluation que o nosso intrepetador faz.
Torna-se, então, imperativo que haja um parser capaz de formar esta àrvore de sintaxe. Num primeiro momento tentou-se construir a àrvore de uma forma um pouco ingénua. Depois de alguma pesquisa persebeu-se o que teria de ser feito para que a àrvore resultá-se. Destaque para a seguinte fonte que explica em detalhe os passos para fazer um simples parser.

O parser funciona com a seguinte lógica: sempre que se encontra digitos, letras ou ^ é considerado que se está perante um polinómio e portanto são mapeadas todas as strings que obdecem a essa regra com a função `parseExpr` que se responsablisa por chamar a função `parsePoli` e de trasformar o seu output numa expressão terminal. Caracteres como `+`, `-`, `*`, `'` (derivada) e `**` são entendidos como operadores aos quais chamamos a Expr corresponde. Este são depois encandeados com outras expressões respeitando sempre a prioridade de operadores, ver `expr` e `subexpr` no ficheiro Parser.hs.

Normalizar Polinómio

A normalização de polinómios passa por analisar os monómios que o constituiem. Um polinómio normalizado deve ter os seus monómios ordenados descendentemente por grau e variáveis que possuem.
Para tal, recorre-se a funções específicas de sorting que recorrem a `sortBy` pelas ordens e critérios que desejamos.

```
-- | Checks the monomio greater. First check the exponents and then the variables
monoSort :: Monomio  -- ^ Monomio
         -> Monomio  -- ^ Monomio
         -> Ordering -- ^ True if left is greater than the right monomios exponents
monoSort a b | greatExpba = GT -- If the variables are equal then we want sorted with descending order
             | greatExpab = LT
             | snd a > snd b  = GT
             | snd a < snd b  = LT
             | otherwise = LT
                where greatExpba = checkGreaterExp (monoExp b) (monoExp a)
                      greatExpab = checkGreaterExp (monoExp a) (monoExp b)
```

Deve-se também assegurar que não temos monómios com coeficiente nulos (estes são removidos). É igualmente necessário ter em atenção os expoentes nulos que levam à remoção da variável em questão.

Somar Polinómios

Começando pelo caso mais simples, a soma de dois monómios é apenas possível caso estes possuem iguais variáveis com os mesmos expoentes. O resultado desta operação resulta em somar os coeficientes e manter as variáveis e respetivos expoentes.

A soma de dois polinómios nada mais é do que a concatenação de dois polinómios `([Monómio] ++ [Monómio])` seguida de uma tentativa de juntar todos os monómios que obedeçam ao critério de soma de monómios acima mencionado.

Para facilitar o processo, após ter uma única lista de monómios, normalizamo-la (operação explicada anteriormente) para garantir que monómios possíveis de serem somados estarão adjacentemente. Depois trata-se apenas de fazer chamadas recursivas que comparam cada par adjacente de monómios e, se compatíveis, os substitui pelo resultado da sua soma.

Multiplicar Polinómios

Partindo do caso mais simples, isto é, a multiplicação de dois monómios, podemos depois extrapolar para o caso de polinómios.
A multiplicação envolve uma multiplicação simples dos coeficientes seguida de uma verificação das variáveis e respetivos expoentes. Para tal, fazemos uma string de variáveis que seja o set das strings de cada um dos monómios e depois vamos a cada monómio ver que expoente corresponde a cada uma destas letras para somarmos os expoentes no resultado final.

Para multiplicar polinómios, basta aplicar uma operação distributiva a cada combinação de monómios que estes polinómios possuam.

Derivar Polinómios

A derivada de polinómios é bastante simples. Trata-se apenas de aplicar uma função de derivar monómio a cada elemento usando um `map`.
Para derivar um monómio temos duas situações possíveis. Caso a variável em ordem à qual estamos a derivar não esteja presente na string de variáveis, o resultado é automaticamente zero.
Caso contrário, basta procurar o expoente dessa variável, subtrair-lhe 1 e multiplicar o antigo expoente pelo coeficiente do monómio.

Parsing Output

O parsing da representação interna para string envolve, à semelhança de todas as operações anteriores, uma conversão de cada um dos monómios para string seguida de uma função para juntar cada uma dessas strings, obtendo, assim, o polinómio a ser printado.

Na passagem da representação interna escolhida para string, temos de ter em atenção várias coisas:

- o coeficiente pode ser nulo (ignorar termo) ou 1 (omitir coeficiente)
- o expoente pode ser 1 (omitir expoente)
- podemos estar perante um monómio simples, multivarável ou até mesmo perante uma constante

Para um output mais correto e organizado, é necessária a aplicação prévia de uma normalização ao polinómio a ser convertido para string.

Funcionalidades Extras

Foram implementadas três funcionalidades que auxiliam ao usar o programa. São elas `**` , `!` e `-`.
A primeira representa operação de potência para polinómios. A segunda serve para representar a operação feita anteriormente, acabando por funcionar como um "ANS" numa calculadora comum. A terceira corresponde à operação de subtração de polinómios.

Exemplos de utilização

```
> 3x^3 - 5y -5x^3                    # Soma simples
- 2x^3 - 5y

> (5x^3 + 3y) * (2z - 4w^3)          # Multiplicação com distributiva
- 20w^3*x^3 - 12w^3*y + 10x^3*z + 6y*z

> 1+2*3                              # Precedência de operações com constantes
7

> -(2y^2 -3z)                        # Negativa de parentesis
- 2y^2 + 3z

> 3x*4y^3+5z                         # Precedência de operações com variáveis
12x*y^3 + 5z

> (3x^6)'x                           # Derivada Simples
18x^5

> (3x^6 + 4y^3 -7x^2z^4)'x           # Derivada Com Multiplas Varíaveis
18x^5 - 14x*z^4

> (2y^2 -3z)**2                      # Operação de potência
4y^4 - 12y^2*z + 9z^2

> 5x + 3y -2x -3z^2                  # Outro exemplo de soma
- 3z^2 + 3x + 3y

> !*(4y^7)                           # Uso do operador "!" que usa o resultado anterior
12y^8 - 12y^7*z^2 + 12x*y^7

> (5x * (3y^2 -2z^3) - 2) * (4x^3)'x  # Exemplo de nested parentesis com multiplas operações
- 120x^3*z^3 + 180x^3*y^2 - 24x^2

> ((x+1)**2)'x                       # Operação de potência de polinómio seguido de uma derivada
2x + 2
```

Grupo

- Marco André (up202004891)
- Ricardo Matos (up202007962)

23/10/2022