Functional and Logic Programming

Developed an Haskell module to work with polynomials including the following features: normalize, add, multiply, derivate, output and input polynomials.

Instructions

To run the program use the `ghc` or the run either the `main` or `cycler` functions with the interpreter. This will prompt an interactive program that allows the use of the functionalities mentioned above.

Using the interpreter, it is possible to use individual functions, which we list below.

Polynomial Representation

A polynomial is a set of monomials. In turn, a monomial is composed of a coefficient and variables accompanied by exponents.

Bearing this in mind, the chosen representation for the variables was `(String, Int)`. The first field would be the string name and the second one it's power. `x^2 = ("x", 2)`. In turn, to create a monomial we just need a list of variables and a coefficient: `(Int, [Variable])`. Finally, a polynomial will be a list of monomials(`[Mon]`), as such: `2x^2 + y = [(2, [("x", 2)]), (1, [("y", 1])]`.

Relevant functionalities

Input

After the internal representation is set, we now need to create a way to transform an input string to a `Pol` type. This can be done using the `toPol` function. In its implementation, it parses a monomial at a time and divides it even further by parsing the numbers and variables separately. The general idea is to read a number/variable until a delimiting char is found (e.g. `*`) and construct the polynomial upon this.

Output

To output a `Pol`, `printPol` can be used. The idea behind it is similar to the input function. It moves its way until the end of the polynomial and prints all its elements in a readable form.

Normalize

To normalize a polynomial, `normalize` can be used. The idea behind it is to first group the polynomial in to like terms (monomials with the same variables and exponents) using `likeTerms` and then add these up together. In turn, `likeTerms` works with the help of the `groupBy` function, which will group like terms together using a provided comparison function. Note that for this to work the polynomial should be previously sorted according to specific factors, which is done by `sortPol`.

Addition

Addition of polynomials can be done with `add`. Because a polynomial is a list of monomials, this function simply groups the monomials in a single list and calls `normalize`, which will do the additions by itself.

Multiplication

Multiplying polynomials can be done with `multiply`. The idea behind it is using list comprehension to combine the monomial lists into pairs and use `monMultiply` with the results. The latter works by multiplying the coefficients and joining the variable lists together. It then resources to `simplifyMon`, that groups matching variables and adjusts exponents accordingly.

Derivation

Last but not least, the derivation of a polynomial can be done with `derivate`. As an example the derivation of a monomial `c*v^e` in order of v would translate to `(c * e) * v ^ (e - 1)`. The implemented function uses the exact same logic and does it to each monomial in the polynomial.

Examples

Example 1

As previously stated, an interactive menu can be prompted by the use of `main` or `cycler`:

```
*Polynomial> main

Choose an option:
a) normalize a polynomial
b) add polynomials
c) multiply polynomials
d) derive a polynomial
e) exit
Choice:a

Non normalized polynomial: 0*x^2 + 2*y + 5*z + y + 7*y^2
Normalized polynomial:
7y^2 + 3y + 5z
```

Example 2

Individual can also be access without the need for the interactive menu, as such:

```
*Polynomial> multiply "4x*y + 5x^2" "x + 2"
4x^2*y + 8x*y + 5x^3 + 10x^2

*Polynomial> derivate "x" "4x^2*y + 8x*y + 5x^3 + 10x^2"
20x + 15x^2 + 8y + 8x*y

*Polynomial> add "4time*position + 10"  "velocity^2 + 3*position"
velocity^2 + 10 + 3position + 4time*position
```

NOTE: Because variables can also be strings, * should be used to separate variables without exponents.

Example 3

It is also possible to work with the internal representation of the polynomials.

```
*Polynomial> a = toPol "4x + 2x*y"
*Polynomial> a
[(4,[("x",1)]),(2,[("x",1),("y",1)])]

*Polynomial> b = polDerivate "y" a
*Polynomial> b
[(2,[("x",1),("y",0)]),(0,[("x",1)])]
*Polynomial> printPol b
2x
```

About

Developed by:

- Igor Diniz *up202000162*
- José Luís Rodrigues *up202008462*

October 2022