PFL_TP1

Polynomial manipulation

Requirements

- ghc 9.0.1
- cabal 3.4.0.0

With ghcup installed, all that is needed is to execute

```
ghcup install ghc 9.0.1
ghcup set ghc 9.0.1
ghcup install cabal 3.4.0.0
ghcup set cabal 3.4.0.0
```

QuickStart

```
cabal update
cabal install
cabal build
```

Usage

Running the user interface

```
cabal run app
```

6 options will be provided corresponding to the 5 functionalities and exiting the program. Type one of those 6 numbers according to your needs and press enter. Afterwards, it will ask for a polynomial. Type the wanted polynomial and press enter. It may additionally ask for a variable or another polynomial. Type that and press enter. The result will be output to the screen and the 6 options will be shown again. This will continue happening until you choose the option to exit the program.

Testing

```
cabal test
```

Description

This project consists of a program that manipulates polynomials. The program is able to read polynomials via user input, perform operations on them, transform them back to strings and then output them. The operations that can be performed are: - Normalization - Addition - Multiplication - Derivation

Internal representation

We thought of two different representations:

Array of Monomials

each being a tuple of a coefficient and an array of variables and exponents

Example

Polynomial: 4x ^2y + z w^3 Internal representation: [(4,[('x',2),('y',1)]),(1,[('z',1),('w',3)])]

A n-dimensional array

n being the number of different variables, where for each possible combinations of exponents from the minimum to the maximum exponent of each variable the corresponding coefficient is stored.

Example

Polynomial: xy^2 + x^2 + xy + 2y +3

	0	1	2
0	3	2	0
1	0	1	1
2	1	0	0

Where the vertical exponents correspond to the variable x and the horizontal exponents corresponds to the y variable.

Our choice

The first representation needs less space, but the operations are more complex. For example, with the second representation addition is just adding the coefficients of the two n-dimensional arrays. However, using the second representation, the simple polynomial x^32yz would need to store 32^3 = 32768 coefficients. Clearly this representation would be incredibly inefficient for sparse polynomials with more than a few variables. We chose the first representation due to it's flexibility.

Parsing

- First, we filter characters needed to represent the polynomial, removing unnecessary characters (spaces, the multiplication symbol, etc) and add a plus sign in the begging if it is omitted (we need it to parse monomials). After this, we iterate the string, skipping a character every time we find a carat ('^') (in order to ignore characters representing the sign of exponents) and add a separator ('?') before each coefficient signal. Afterwards, we use this separator to split the input string into a list of strings representing monomials. Finally, we parse each monomial.
- To parse a monomial, we use the first character to get the sign of the coefficient and takeWhile isDigit to get it's absolute value. Afterwards, we recursively iterate the rest of the monomial. The second character will always be a carat (^), unless the exponent is omited. Hence if it isn't, then we can store that variable with exponent one on our internal representation. If it is, we can just takeWhile isPartOfNumber (isDigit || '+' || '-') in the string starting after the carat to get the exponent of the variable. Either way, we recursively call the function with the rest of the string. In the end, if there is only one character in the string then that character is a variable with exponent one, ommited.

Normalization

To normalize polynomials, we first transverse each monomial one variable at a time, removing variables with coefficient zero . After this, we join equal monomials (e.g. xy^2 + 2xy^2 becomes 3xy^2). We do this by iterating the polynomial one monomial at a time and, for each monomial, transversing the rest of the list and summing the coefficients of monomials with the same variables and exponents as the one considered in the current iteration. We add the result of this sum to the coefficient of the current monomial and remove the monomials whose coefficients contributed to the sum (which have equal variables and corresponding exponents in relation to the current monomial). After this, we remove monomials with coefficient zero (zero product rule: 0xy = 0) and then order the polynomial (monomials with a higher degree first).

Addition

This function assumes the variables are already alphabetically sorted (which we guarantee they are before and after each operation) Adding two polynomials is the same as normalizing a polynomial consisting of all the monomials in both polynomials.

Hence, we joined the two lists and called the function which normalizes.

Multiplication

We used a list comprehension to apply the distributive law of multiplication over addition. For each resulting pair of monomials, we multiplied them using another function. This function multiplies the coefficients and joins the variables (first just appends one list of variables to the other, but then we normalize the variables by iterating one variable at a time and, for each variable, transversing the rest of the list and summing the exponents of variables equal to the one considered in the current iteration. We add the result of this sum to the exponent of the current variable and remove the variables whose coefficients contributed to the sum (which are equal to the current variable). Afterwards we reorder the variables alphabetically and normalize the resulting polynomial.

Derivative

We iterate the list of monomials and, for each monomial, we get the exponent of the variable being derived by (by iterating the variables and comparing the current variable to the one being derived by). If the result of this function is 0, then the variable isn't present in the monomial, and the derivative of that monomial is 0. If it isn't then we multiply the coefficient of the monomial by the exponent we found and subtract one to the exponent of the variable being derived by in the list of variables (we append the list of variables without the one being derived by to the variable being derived by with the previous exponent minus 1 and then alphabetically order the result).

Examples to test functionalities

Besides the automatically generated property-based tests, we also used the following examples:

Parsing

- input: "x*y^(+2) + 3*y^(-3)"
- internal representation: [(1,[('x',1),('y',2)]),(3,[('y',-3)])]
- input: "xyz^3 -3xy^-3 +5"
- internal representation: [(1,[('x',1),('y',1),('z',3)]),(-3,[('x',1),('y',-3)]),(5,[])]

Normalize

- polynomial: 0*x^2 + 2*y + 5*z + y + 7*y^2
- result: 7y^2 + 3y + 5z

Sum

- first polynomial: xy + x^2 + z
- second polynomial: -yx +3z + y^2
- result: x^2 + y^2 + 4z

Multiplication

- first polynomial: xy - zx^3
- second polynomial: 0yx + 1xz + yx
- result: -x^4 z^2 - x^4 yz + x^2 y^2 + x^2yz

Derivative

- polynomial: yz^3 + y^2 + z^2 + x^2y + x^-4z - 7x + 5
- variable: x
- result: 2xy - 4x^-5z - 7

Developed by: - Pedro Manuel da Silva Gomes (up202006322@edu.fe.up.pt) - João Rola Reis (up202007227@edu.fe.up.pt)