Students:
Raducanu Alexandru – up202202991
Serban Emilia-Bianca – up202202992

PROJECT

- **Intern representation of a polynomial:**

  We chose to represent a polynomial as a sorted list of **Termen**. Termen is a new data type created by us. It is composed of two fields:
  1. **coeficient** which is of type Double
  2. **text** which is of type String, modelling the variable of that Termen

  Besides the implicit constructor of class Termen which is
  Termen :: Double ->String ->Termen, we created a new constructor
  createTermen :: Double ->String ->Termen which makes sure that the String is alphabetically sorted.

  Our internal representation, although very inefficient regarding storage space, provides more flexibility and ease of implementation for other functionalities. Here are some examples of regular terms of polynomials and their representations in our program:
  - "0" -> Termen (0.00 , "")
  - "12xy" -> Termen (12.00, "xy")
  - "-3x^5y^6a^2" -> Termen (-3.00, "xxxxxyyyyyyaa")

  We also made the Termen class an instance of the Eq, Ord and Show classes, not by deriving them at the definition of the new class, but by implementing "==", "compare" and "show" functions for the Termen class:
  - two Termens are equal when their "coefficients" and "texts" are identical
  - Termens are ordered lexicographically by their "texts", or, in the case of identical texts, they are ordered by the value of the "coefficient"
  - show function transforms a Termen into a String, for instance: show(Termen 42, "xxxyyyy") => " +42x^3y^4"

- **Parsing functionality - fromString :: String -> Polinom**

  We implemented the function "fromString" which receives a String and returns a Polinom. The String must be a valid input, as it is almost impossible to detect all the things that could go wrong in a String, regarding the transformation into a Polinom. That is why we could not implement a function that checks the validity of a String. So, there are the rules that the String should follow:

- The string can only contain '+' , '-' , '^' , letters, spaces and digits
- The String can never use "*" for multiplication ("2*x*y" --→ "2xy")
- The coefficient of a term should always be the first ("2xy", not "xy2")
- Coefficients must be integers, exponents must be positive integers
- Other characters are forbidden

Any violation of these rules can make the program useless, There are multiple and diverse way of violations you can make in a String, so it is very hard to implement a validation logic.

Assuming that the String given as an input is correct, the function deletes all the spaces and splits the String in multiple smaller Strings that represents Termens, using "+" and "-" as delimitators. The smaller Strings are, then, transformed in regular Termens. All of these Termens are concatenated in a list and the list is sorted using the implicit sort() function. Now, we have our Polinom as a sorted list of Termens.

- **Normalizing a Polinom - normalizePolinom :: Polinom -> Polinom**

The Polinom that is created from parsing a valid String is not yet a normalized Polinom. Being a normalized Polinom means that:

- There are no Termens that have the coefficient 0
- There are no multiple Termens with the same "text"

The function "normalizePolinom" receives as an input a Polinom (that is a sorted list of Termens) and takes advantage of the sorting in the process of "normalizing" that Polinom.

- **toString :: Polinom -> String**

This function receives a Polinom and transforms it into a String that can be showed on console. The difference between "toString" and the implicit implementation of "show" (as Polinom is an instance of Show class) is that "toString" gets rid of the "[ ]" and the null Polinom is converted to "0" , not empty list "[]".

- **Sum - addPolinoms :: Polinom -> Polinom -> Polinom**

This function that adds 2 Polinoms takes advantage of the fact that Termens are sorted in both Polinoms. It is very similar to merging 2 sorted lists.

The function uses recursion to "consume" the smallest Termen of the 2 Polinoms at every iteration. If the smallest Termen of Polinom1 and the smallest Termen of

Polinom2 have the same “text”, then we create a new Termen with the sum of the “coefficients”. We continue until one of the Polinoms is fully “consumed”, which is a base case.

- **Multiplying 2 polinoms - multiplyPolinoms :: Polinom -> Polinom -> Polinom**

  First, we created a function that multiplies 2 Termens. Using that function, we created another one, that multiplies a Termen with a Polinom. We need to make sure that the resulted Polinom is sorted, as the multiplication with a Termen can affect the order of the resulted Termens.

  The main function, “multiplyPolinoms” uses the function described above. It “consumes” a Polinom, Termen by Termen, and, each iteration, multiplies the Termen with the second Polinom. The use of function “addPolinoms”, rather than simple concatenation, will assure us that the result is sorted.

  Also, if the 2 input Polinoms are normalizes, the output is for sure a normalized Polinom.

- **Derivative of a Polinom - derivatePolinom :: Char -> Polinom -> Polinom**

  We implemented a function that derivates a Termen with respect to a specific variable. As a Polinom is just a sum of Termen, we “mapped” the function described above on all the Termens of the Polinom.

  After the mapping, we can not be sure that the resulted Polinom is sorted and normalized, so we apply those functions also.

- **Input / Output**

  We also created 4 functions that uses the IO() monad in order to interact with input from the user. Each function illustrates a functionality.

  Because we cannot verify if the String given by the user is valid (meaning that it respects all the rules stated in the **Parsing** section) , there are no errors threw by these IO() functions. So, in order to make sure that these functions work properly, the user has to be very careful with the input he/she provides.

Students:
Raducanu Alexandru – up202202991
Serban Emilia-Bianca – up202202992

- **Module Polinom and Testing**

  As there are a lot of functions and code on a single file, we decided to create the module “Polinom.hs” that contains all of our work. Only the relevant functions will be exported from this Module.

  To check and test our functions, we imported this module in a new file, named “principal.hs” that has some Polinoms already generated. You just need to “load” the “principal.hs” file.

  We mention that we tried using the module Test.QuickCheck, but we could not manage to make “Termen” an instance of the Arbitrary class, in order to use “quickcheck” on some functions. Also, it is difficult to specify the generators for Termens and Polinoms, as there are many conditions a String needs to verify in order to be a valid Polinom (see section **Parsing**).

  For checking personal input without the IO() , define your String in the “principal.hs” file as a variable and then apply (normalizePolinom $ fromString) on the variable. Now, you have a normalized Polinom. Create how many of these you wish and test our functions.