CSE 341, Winter 2009, Assignment 2
Haskell Project
Due: Wednesday Jan 21, 10:00pm

The purpose of this assignment is to give you experience with writing a more realistic program in Haskell, including working with user-defined types, unit tests, and input-output in a purely functional language.

40 points total (10 points per question); up to 10% extra for the extra credit question.

Don't even think about working on the extra credit part until you've completed the main assignment. Do it for what you'll learn, not for the extra points (which are rather small compared with the amount of extra work).

You can use up to 4 late days for this assignment.

For each top-level Haskell function you define, include a type declaration. You should specify unit tests for each of your top-level functions (except for the monadic functions) using the `HUnit` package — so you need unit tests for `poly_multiply` and `show` for Polynomials. The questions below suggest tests to include. Bundle all of your unit tests into an instance of `TestList` called `tests` to make them easy for the TA to run.

All of your functions, except for your interactive function `poly_calc` for Question 3 (and for the extra credit part if you're doing IO), should be written in a pure functional style (no monads or `do` statements).

Turnin: Ideally, turn in a single listing of your program that includes answers to all the questions. (However, if more convenient, for example if you redefine some functions in your polynomial program to make a version that works with infinite polynomials, you can turn in several files.) Also turn in sample output for Question 3 and (if appropriate) 5. You don't need to turn in sample output for Questions 1, 2, and 4 — the unit tests are enough for those. As before, your program should be tastefully commented (i.e. put in a comment before each function definition saying what it does). Style counts! In particular, think about where you can use pattern matching and higher order functions to good effect to simplify your program; and avoid unnecessary repeated computations.

1. Write and test a Haskell function `poly_multiply` that multiplies two polynomials in a symbolic variable and returns the result. The polynomials should be represented as lists of tuples, where each tuple represents a term (a coefficient and an exponent of the symbolic varible). If we define some convenient type synonyms, then its type is as follows:

   ```
   {- a Term is a type synonym for a tuple of a Double and an Integer -}
   type Term = (Double,Integer)
   type Terms = [Term]

   poly_multiply :: Terms -> Terms -> Terms
   ```

   You can assume the terms will be normalized, so that they are sorted with the largest exponent first and that there will be no terms with a coefficient of 0. Also assume that the exponents will be non-negative. The zero polynomial is represented as a polynomial with an empty list of terms. The result should be normalized as well (i.e. drop terms with a coefficient of 0, and keep the terms sorted by exponent).

   For example:

   ```
   p1 =  [(1.0, 3), (1.0, 2), (1.0, 1), (1.0, 0)]
   p2 = [(1.0, 1), (-1.0, 0)]
   ```

```
p3 = poly_multiply p1 p2
-- p3 should now have the value [(1.0, 4), (-1.0, 0)]
```

In standard algebraic notation, this represents

$x^4 - 1 = (x^3 + x^2 + x + 1) \cdot (x - 1)$

Here are some other polynomial pairs to test your function on:

$(-3x^3 + x + 5) \cdot 0$

$(x^3 + x - 1) \cdot -5$

$(-10x^2 + 100x + 5) \cdot (x^{9999} - x^7 + x + 3)$

2. Now define a `Polynomial` type to hold the name of the symbolic variable for a polynomial and its list of terms.

```
data Polynomial = Poly Char Terms
                  deriving (Read)
```

Note that `Polynomial` doesn't derive `Show`. Instead, your `Polynomial` type should use a custom `show` function rather than the one that comes from using the `deriving` construct, so that instances of `Polynomial` type print in a nicer way. This function should return a string representing the polynomial in conventional Haskell syntax. Print the terms sorted by exponent, largest first. Omit the coefficient if it's equal to 1, omit the exponent if it's equal to 1, omit the variable entirely if the exponent is 0 (just give the coefficient), and omit the constant if it's equal to 0 and if there are other terms. If the coefficient is negative, use a minus rather than a plus between the terms at that point. For example:

```
show (Poly 'x' [(1.0, 3), (1.0, 2), (1.0, 1)]) =>  "x^3 + x^2 + x"
show (Poly 'x' [(4.0, 3), (-5.0, 0)])  =>  "4.0*x^3 - 5.0"
show (Poly 'x' [(10.0, 0)])  =>  "10.0"
show (Poly 'x' [])  =>  "0.0"
```

Add appropriate unit tests for your `show` function.

Hint: declare `Polynomial` to be an instance of `Show`:

```
instance Show Polynomial where
    show (Poly x terms) = .....
```

3. Write an interactive function `poly_calc` that prompts the user for two polynomials, and prints the input polynomials and the result of multiplying them. Use the `read` function for `Poly` to read in each polynomial. (This `read` function is defined automatically by deriving the `Polynomial` type from the `Read` type class.) The zero polynomial in x is entered as "`Poly 'x' []`". `poly_calc` should then ask "`again?`". If the user types in `y`, it should ask for two more polynomials and multiply them, and so on until the user wants to stop. Then it should print `bye!` and exit.

The `poly_calc` function should check for polynomials with different symbolic variables, and if they are different should print a warning, prompt for two new polynomials, and try again. However, you don't need to handle other kinds of bad inputs. (But see the extra credit part.)

If you want to see how the program should work on various inputs, there is a version on attu on `~borning/calc`.

4. Make up, write, and test your own Haskell script that uses infinite data structures in an interesting way. Remember to include unit tests for these. If you can't think of anything interesting, make up a program that uses them in an uninteresting way ... you don't need to do anything big to get full credit for this question, but I'm hoping a few students will do something fun with it. One rather challenging possibility would be to allow infinite polynomials in your polynomial multiplier.

5. (extra credit) There are several possibilities for extra credit work on the polynomial multiplier. For all of these you should draw on the extensive Haskell library.

   - Use Haskell's exception handling mechanism to appropriately handle input that doesn't parse correctly as a `Poly` by asking the user to enter the polynomial again. Also allow zero coefficients and negative exponents: zero coefficients should be just dropped from the polynomial, and negative exponents should be supported correctly.
   - Produce nicely formatted output, for example in html.
   - Write a GUI to supplant the interactive `poly_calc` function.