CSE 341, Winter 2010, Assignment 2
Haskell Project
Due: Wednesday Jan 20, 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.

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.

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 terms, where each term consists of a coefficient and an exponent of the symbolic variable. If we define some convenient types and type synonyms, then the type of `poly_multiply` is as follows:

   ```
   data Term = Term Integer Integer
               deriving (Read,Show)

   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, there won't be two terms with the same exponent, 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 =  [Term 1 3, Term 1 2, Term 1 1, Term 1 0]
   p2 = [Term 1 1, Term (-1) 0]
   p3 = poly_multiply p1 p2
   -- p3 should now have the value [Term 1 4, Term (-1) 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 String Terms
                     deriving (Read)
   ```

   We include `deriving (Read)` here mostly to help in debugging – in Question 3 we'll make use of a parser to parse polynomials entered in normal Haskell notation.

   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 (since the polynomial is normalized, the terms will already be in this order in the list). 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. Finally, print "0" for the zero polynomial. For example:

   ```
   show (Poly "x" [Term 1 3, Term 1 2, Term 1 1]) =>  "x^3 + x^2 + x"
   show (Poly "x" [Term 4 3, Term (-5) 0])  =>  "4*x^3 - 5"
   show (Poly "x" [Term 10 0])  =>  "10"
   show (Poly "x" [])  =>  "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, prints each input polynomial back in normalized form, and then prints the result of multiplying them. It continues prompting for more polynomials until the user types a blank line for the first polynomial.

   The user should be able to enter the polynomials in standard Haskell syntax (the same as used for the output from `show` in Question 2). Since writing a robust parser is probably more than is reasonable to ask for on this assignment, we will give you a package `PolyParser.hs` that defines an appropriate parser, generated by the Haskell "Happy" parser generator. This is linked from the assignment page. To use PolyParser, download the file and put it in the same directory as your own program. Then include this statement in your own program:

   ```
   import qualified PolyParser
   ```

   We're using `qualified` to avoid name collisions with functions and datatypes defined in PolyParser. The function you want from this module is `string_to_raw_poly`, which has the following type:

```
string_to_raw_poly :: String -> (String,[(Integer,Integer)])
```

This function takes a string representing a polynomial, parses and normalizes it, and returns a representation of the raw polynomial as a String (the variable) and a list of coefficient-exponent pairs. This is already normalized (the terms are sorted and combined if need be; terms with a 0 coefficient are dropped from the list). Note that for constants (e.g. `10`) the function can't figure out the name of the variable, so it will use the empty string in that case.

So most of the error checking is already done for you by the parser — the only other error you need to check for is trying to multiply two polynomials in different variables.

There is a compiled version of this program on attu that you can try if you have questions about how it should behave — to run it, just type this at the unix prompt:

```
~borning/poly
```

Try your program on the test cases above. Here are a few others to try (and reasons for them):

- `3` times `x` (make sure it knows that the variable is `x`)
- `x` times `3` (similarly)
- `x^2` times `0` (make sure 0 is working)
- `0` times `x^2` (similarly)

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 problems. There are many possibilities for extra credit work on the polynomial multiplier. There are two suggestions below; but if you think of something else you're interested in, please ask.

   - Write a GUI to supplant the interactive `poly_calc` function.
   - Allow negative exponents. (This is a bit tricky since you'll need to modify the parser — start with the `PolyParse.y` file, modify it, and run `happy` to generate a new parser.) There is information on `happy` at `http://www.haskell.org/happy/`. It doesn't seem to be pre-installed on the lab machines, but it's included in the Haskell Platform, so you can install it easily on a personal machine, or on one of the lab machines if you install Haskell yourself.