CSE 341, Autumn 2008, Assignment 2
Haskell Project
Due: Friday October 10, 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 3 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. 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).

Turn in a single listing of your program (that includes answers to all the questions). 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.

1. Write and test a Haskell function `poly_multiply` that multiplies two polynomials in a symbolic variable and returns the result. You should define a convenient representation for the polynomials, but it must employ one or more user-defined types. Here is one suggestion for a starting point for types to use — you will need to augment this a bit to run the unit tests. (You can also do something different if you prefer.)

```
{- a Term has a coefficient and an exponent -}
data Term = Term Double Int
            deriving (Show)
{- a symbolic polynomial has the name of the symbolic variable,
   and a list of Terms in the polynomial -}
data Polynomial = Poly Char [Term]
                  deriving (Show)
```

   The result should be simplified, i.e. drop terms with a coefficient of 0. For example:

```
p1 = Poly 'x' [Term 1.0 3, Term 1.0 2, Term 1.0 1, Term 1.0 0]
p2 = Poly 'x' [Term 1.0 1, Term (-1.0) 0]
p3 = poly_multiply p1 p2
-- p3 should now have the value Poly 'x' [Term 1.0 4, Term (-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:

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

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

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

Hints: you can find the source code and documentation for the `HUnit` package at `http://hunit.sourceforge.net/`. There is also a copy of the code on the undergrad linux machines at `/cse/courses/cse341/common/HUnit-1.0`.

Here is a minimal example of using `HUnit`. Suppose you've got a file `UnitTestExample.hs` with this contents:

```
module Main
    where

import HUnit

test1 = TestCase (assertEqual "arithmetic test" (3+4) 7)
tests = TestList [TestLabel "test1" test1]
runtests = runTestTT tests
```

Here we've defined a unit test `test1` that checks whether 3+4 is equal to 7. `tests` is a list of unit tests. `runtests` runs the tests. To run this, Haskell needs to know about `HUnit`. The simplest way to do this is to copy all of the HUnit files into the same directory in which `UnitTestExample.hs` is living, although this clutters up your directory.

Alternatively, you can tell Haskell where `HUnit` is living using the `-i` option to ghci. On attu or another CSE linux machine, you can use the copy in the shared directory:

```
ghci -i/cse/courses/cse341/common/HUnit-1.0/ UnitTestExample.hs
```

On other machines, replace the argument to `-i` with the path to whereever you put the `HUnit` files. On the CSE ugrad windows machines, you will need to run ghci from the command line to use the -i option. (Or just copy the files if you want to continue starting it from the Start menu.)

If you want to compile an executable, do this:

```
ghc -i/cse/courses/cse341/common/HUnit-1.0/ --make UnitTestExample.hs -o test
```

Then `test` should be an executable version of your program.

2. Change your `Polynomial` type to 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' [Term 1.0 3, Term 1.0 2, Term 1.0 1])  =>  "x^3 + x^2 + x"
show (Poly 'x' [Term 4.0 3, Term (-5.0) 0])  =>  "4.0*x^3 - 5.0"
show (Poly 'x' [Term 10.0 0])  =>  "10.0"
show (Poly 'x' [])  =>  "0.0"
```

Add appropriate unit tests for your `show` function.

Hint: to do this, remove the `show` from the `deriving` part of the declaration for `Polynomial`, and declare `Polynomial` to be an instance of `Show`:

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

3. Write an interactive function `poly_calc` that prompts the user for the symbolic variable and the coefficients for two polynomials, and prints the input polynomials and the result of multiplying them. You can decide what format to use for entering the polynomials. For example, you could type the coefficients and exponents one at a time, or type in the entire polynomial in one line.

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.

   - Allow the input polynomials to be entered in Haskell syntax. Use an existing Haskell parser package of some sort.
   - Produce nicely formatted output, for example in html.
   - Write a GUI to supplant the interactive `poly_calc` function.