Functional Programming Skills
Assignment 3

Arthur Nunes-Harwitt

Explicitly write the type of each function.

1. (10 points) Recall the following integrals.

$$\sin(x) = \int \cos(x)\, dx + C_1$$
$$\cos(x) = -\int \sin(x)\, dx + C_2$$

   For power series, $C_1 = 0$ and $C_2 = 1$. Use the integrals to define the power series for sine and cosine for both `Double` and `Rational` types. Name them `sinPowSeD`, `cosPowSeD`, `sinPowSeR`, and `cosPowSeR`.

2. (10 points) Given an angle θ of small magnitude, the power series approximation to $\sin(\theta)$ is reasonably good. However, given an angle θ of large magnitude, the quality of the power series approximation quickly deteriorates. It is possible to ensure an angle of small magnitude by interpreting the following trigonometric identity recursively.

$$\sin(\theta) = 3\sin\left(\frac{\theta}{3}\right) - 4\sin^3\left(\frac{\theta}{3}\right)$$

   Use the approximation $\sin(\theta) \approx \theta$ when $|\theta| < 0.000000001$, and use the identity above otherwise. Make sure you only have one recursive call. Call your function `mySine`.

3. (45 points) Consider a system of linear equations.

$$\begin{array}{ccccccc} a_{11}x_1 & + & \cdots & + & a_{1n}x_n & = & b_1 \\ \vdots & & \ddots & & \vdots & & \vdots \\ a_{n1}x_1 & + & \cdots & + & a_{nn}x_n & = & b_n \end{array}$$

   We can represent this system as a matrix as follows.

$$\begin{pmatrix} a_{11} & \cdots & a_{1n} & b_1 \\ \vdots & \ddots & \vdots & \vdots \\ a_{n1} & \cdots & a_{nn} & b_n \end{pmatrix}$$

Which we can then represent in Haskell as a list of lists.

$$\begin{bmatrix} \begin{bmatrix} a_{11}, & \cdots & a_{1n}, & b_1 \end{bmatrix}, \\ \begin{bmatrix} \vdots & \ddots & \vdots & \vdots \end{bmatrix} \\ \begin{bmatrix} a_{n1}, & \cdots & a_{nn}, & b_n \end{bmatrix} \end{bmatrix}$$

This problem involves writing functions to solve a system of linear equations represented as above. Your functions should allow the most general numeric type; you will find the type classes `Num` and `Fractional` useful.

(a) Write a function `sub` to subtract two lists.
Example:
`sub [2,2,3] [2,5,12]` ⇝ `[0,-3,-9]`

(b) Write a function `scaleList` that multiplies the elements of a list by a constant.
Example:
`scaleList (1/2) [2,5,12]` ⇝ `[1.0,2.5,6.0]`

(c) Write a function `subScale` that scales the first list so that when it is subtracted from the second, the first number in the result is zero. Since it must be zero, return a list without that first constant.
Example:
`subScale [2,2,3,10] [2,5,12,31]` ⇝ `[3.0,9.0,21.0]`

(d) Write a function `nonZeroFirst` that takes a lists of lists, and finds the first list of number that does *not* start with a zero. If all the lists start with a zero, then signal an error. The time complexity should be $\mathcal{O}(n)$.
Example:
`nonZeroFirst [[0,-5,-5],[-8,-4,-12]]` ⇝ `[[-8,-4,-12],[0,-5,-5]]`

(e) Write a function `triangulate` that takes a list of lists representing a system of linear equations, and returns a triangular list representing the same system. It should repeatedly invoke `subScale`; the function `nonZeroFirst` will also play an important role.
Example:
`triangulate [[2,3,3,8],[2,3,-2,3],[4,-2,2,4]]` ⇝
`[[2.0,3.0,3.0,8.0],[-8.0,-4.0,-12.0],[-5.0,-5.0]]`

(f) Write a function `dot` that computes the dot product of two lists.
Example:
`dot [1,2] [3,4]` ⇝ `11`

(g) Write a function `solveLine` that takes a list representing an equation, and a list representing the values for all the variables but the first, and returns the solution for that equation.
Example:
`solveLine [2,3,3,8] [1,1]` ⇝ `1.0`

(h) Write a function `solveTriangular` that takes a triangular list of lists and returns the list of solutions for the system.
Example:
`solveTriangular [[2.0,3.0,3.0,8.0],[-8.0,-4.0,-12.0],[-5.0,-5.0]]` ⇝ `[1.0,1.0,1.0]`

(i) Write a function `solveSystem` that takes a rectangular lists of lists representing a system of equations and returns the list of solutions for the system.
Example:
`solveSystem [[2,3,3,8],[2,3,-2,3],[4,-2,2,4]]` ⇝ `[1.0,1.0,1.0]`

4. (10 points) Write a more elaborate show function for expressions that puts parentheses only where necessary. To do so, modify the code involving expressions to handle rational numbers rather than integers, change the name of the numeric constructor from `IExp` to `RExp`, and write the following five functions.

   - `addParens`, which takes a string and returns a string with parentheses added.
     Example:
     `addParens "x"` $\rightsquigarrow$ `"(x)"`
   - `showSumContext`, which takes an expression and returns a string representing the expression assuming the context is a sum.
   - `showProdContext`, which takes an expression and returns a string representing the expression assuming the context is a product.
   - `showPowContextLeft`, which takes an expression and returns a string representing the expression assuming the context is a left sub-expression of a power.
   - `showPowContextRight`, which takes an expression and returns a string representing the expression assuming the context is a right sub-expression of a power.

   Finally, when creating the instance of `Show`, let `show` be defined by `showSumContext`.

   Examples:

```
*Assign4> (Sum (RExp 2) (Sum (RExp 3) (RExp 4)))
2+3+4
*Assign4> (Sum (Sum (RExp 2) (RExp 3)) (RExp 4))
2+3+4
*Assign4> (Sum (RExp 2) (Prod (RExp 3) (RExp 4)))
2+3*4
*Assign4> (Prod (Sum (RExp 2) (RExp 3)) (RExp 4))
(2+3)*4
*Assign4> (Prod (RExp (2%3)) (RExp 3))
2/3*3
*Assign4> (Pow (RExp (2%3)) (RExp 3))
(2/3)^3
*Assign4> (Pow (Sum (RExp 1) (RExp 3)) (RExp 2))
(1+3)^2
*Assign4> (Pow (Prod (RExp 4) (RExp 3)) (RExp 2))
(4*3)^2
*Assign4> (Pow (Prod (RExp 4) (RExp 3)) (Sum (RExp 2) (RExp 4)))
(4*3)^(2+4)
*Assign4> (Pow (RExp (-1)) (RExp 2))
(-1)^2
*Assign4> (Pow (Pow (RExp 2) (RExp 3)) (RExp 2))
(2^3)^2
```

```
*Assign4> (Pow (RExp 2) (Pow (RExp 3) (RExp 2)))
2^3^2
```

5. (10 points) First, declare a data structure `Eqn` (using the constructor of the same name) that has as parts two expressions. Using the functions to convert to multivariate polynomials, write a function `system` that takes a list of equations and returns a list of the form expected by `solveSystem` above. If the system is non-linear, generate an error. HINT: `KVars` are ordered. You may find it useful to do sorting. The function `sortBy` can be imported from the module `Data.List`.

   Examples:

```
*Assign4> system [(Eqn (Prod (RExp 2) (Var "y")) (RExp 10))]
[[2 % 1,10 % 1]]
*Assign4> system [(Eqn (Var "y") (RExp 5)),
                  (Eqn (Sum (Var "x") (Var "y")) (RExp 2))]
[[1 % 1,1 % 1,2 % 1],[0 % 1,1 % 1,5 % 1]]
*Assign4> system [(Eqn (Var "y") (RExp 5)),
                  (Eqn (Sum (Var "x") (RExp 1)) (RExp 2))]
[[1 % 1,0 % 1,1 % 1],[0 % 1,1 % 1,5 % 1]]
*Assign4> system [(Eqn (Sum (Prod (RExp 2) (Var "x"))
                            (Sum (Prod (RExp 3) (Var "z"))
                                 (Prod (RExp 3) (Var "y"))))
                       (RExp 8)),
                  (Eqn (Sum (Prod (RExp (-2)) (Var "z"))
                            (Sum (Prod (RExp 3) (Var "y"))
                                 (Prod (RExp 2) (Var "x"))))
                       (RExp 3)),
                  (Eqn (Sum (Prod (RExp (-2)) (Var "y"))
                            (Sum (Prod (RExp 4) (Var "x"))
                                 (Prod (RExp 2) (Var "z"))))
                       (RExp 4))]
[[2 % 1,3 % 1,3 % 1,8 % 1],
 [2 % 1,3 % 1,(-2) % 1,3 % 1],
 [4 % 1,(-2) % 1,2 % 1,4 % 1]]
```

6. (15 points) Implement differentiation in an alternative manner.

   (a) Add a differentiation operator `D` to the expression data structure. `D`'s components should be an expression and a string.

   Examples:

```
*Assign4> (D (Pow (Var "x") (RExp 2)) "x")
D(x^2, x)
*Assign4> (Prod (RExp 5) (D (Pow (Var "x") (RExp 2)) "x"))
5*D(x^2, x)
```

(b) Create a third simplifier based on the second. Add simplification rules for the third simplifier to do the differentiation. Make sure to have a rule for each case that the procedure `deriv` handles; however, the rules should not call `deriv`.

Examples:

```
*Assign4> simplify3 (D (RExp 2) "x")
0
*Assign4> simplify3 (D (Var "x") "x")
1
*Assign4> simplify3 (D (Var "y") "x")
0
*Assign4> simplify3 (D (Sum (RExp 2) (Var "x")) "x")
1
*Assign4> simplify3 (D (Sum (Var "x") (Var "x")) "x")
2
*Assign4> simplify3 (D (Prod (RExp 5) (Var "x")) "x")
5
*Assign4> simplify3 (D (Prod (Var "x") (Var "x")) "x")
2*x
*Assign4> simplify3 (D (Prod (Var "x") (Sum (Var "x") (RExp 3))) "x")
2*x+3
*Assign4> simplify3 (D (Pow (Var "x") (RExp 2))  "x")
2*x
*Assign4> simplify3 (D (Prod (RExp 5) (Pow (Var "x") (RExp 2)))  "x")
10*x
*Assign4> simplify3 (D (Prod (RExp 5) (Pow (RExp 2) (Var "x")))  "x")
5*D(2^x, x)
*Assign4> simplify3 (D (Prod (Var "x") (Pow (RExp 2) (Var "x")))  "x")
x*D(2^x, x)+2^x
*Assign4> simplify3 (Prod (RExp 5) (D (Pow (Var "x") (RExp 2))  "x"))
10*x
```

Graduate Problems/Undergraduate Extra Credit

1. (10 points) Recall that $\sqrt{x} = e^{(1/2)\ln(x)}$. Write another function to compute square roots as follows.

   (a) Use the power series for e^x, the function to convert a power series to a sequence, and the limit function to write a function that approximates e^x. Name this function `myExp`.

   (b) Find a suitable integral to use to define the power series for $\ln(x+1)$. Define this series for both `Double` and `Rational` types. Name them `lnPlusPowSeD` and `lnPlusPowSeR`.

   (c) Write a function `reduce` that takes a `Double` x and returns a pair of `Doubles` $(x/10^n, n)$, where n is the smallest natural number such that $|x/10^n| < 1$. Also define a constant `ln10` to be a `Double` approximating $\ln(10) \approx 2.302585092994046$.

   (d) It turns out that the power series above for $\ln(x+1)$ converges slowly. Instead, we will take an approach similar to our approach with square roots. We will make an initial guess ($g_0 = 1$) and then construct a

sequence of improvements. The improvement is $g_{n+1} = g_n + 2 \times \dfrac{x - e^{g_n}}{x + e^{g_n}}$, where x is the number we are trying to compute the natural log of. This improvement works best when $|x| < 1$, so use the `reduce` function. When computing e^{g_n}, use the function `myExp`.

(e) Write the function `myLn` making use of that sequence. Combine the functions `myExp` and `myLn` so as to implement a function that computes square roots. Name this function `mySqrt'`.

2. (30 points) First, add operators to the expression data structure that represent the functions sine, cosine, e^x, and log. They should be called `Sin`, `Cos`, `Exp`, and `Ln` respectively. Also add the integration operator `Int`. Extend `show`. Add rules to the simplifier so that it is possible to take the derivative of any expression. Also add rules to the simplifier so that simple expressions can be integrated: constants, x^n, $\sin(x)$, etc. And add a rule to turn the integral of a sum into the sum of integrals. Finally, add the following two heuristic product rules.

(a) $\displaystyle\int y \frac{dy}{dx}\, dx = \int y\, dy = \frac{1}{2} y^2$
Example:
$\displaystyle\int \sin(x)\cos(x)\, dx = \frac{1}{2}\sin^2(x)$
This rule can be viewed as a special case of the next rule.

(b) This rule is known as "derivative divides." It is a more algorithmic way to characterize u-substitution. It involves the following steps.

- For $\int y\, dx$, pick a factor of y and call it $f(u)$.
- Compute the derivative du/dx.
- Divide y by $f(u) \times du/dx$. The division may be heuristic, involving merely pattern matching. Call the quotient q.
- If q is a constant, then the result is $q \int f(u)\, du$.

Example:
$\displaystyle\int x \sin(x^2)\, dx = \frac{1}{2}\int \sin(u)\, du = -\frac{1}{2}\cos(x^2)$