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.

$$\begin{array}{cccccc} [& [& a_{11}, & \cdots & a_{1n}, & b_1 \quad], \\ & & \vdots & \ddots & \vdots & \vdots \\ & [& a_{n1}, & \cdots & a_{nn}, & b_n \quad] \quad] \end{array}$$

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:

```
*Assign3> (Sum (RExp 2) (Sum (RExp 3) (RExp 4)))
2+3+4
*Assign3> (Sum (Sum (RExp 2) (RExp 3)) (RExp 4))
2+3+4
*Assign3> (Sum (RExp 2) (Prod (RExp 3) (RExp 4)))
2+3*4
*Assign3> (Prod (Sum (RExp 2) (RExp 3)) (RExp 4))
(2+3)*4
*Assign3> (Prod (RExp (2%3)) (RExp 3))
2/3*3
*Assign3> (Pow (RExp (2%3)) (RExp 3))
(2/3)^3
*Assign3> (Pow (Sum (RExp 1) (RExp 3)) (RExp 2))
(1+3)^2
*Assign3> (Pow (Prod (RExp 4) (RExp 3)) (RExp 2))
(4*3)^2
*Assign3> (Pow (Prod (RExp 4) (RExp 3)) (Sum (RExp 2) (RExp 4)))
(4*3)^(2+4)
*Assign3> (Pow (RExp (-1)) (RExp 2))
(-1)^2
*Assign3> (Pow (Pow (RExp 2) (RExp 3)) (RExp 2))
(2^3)^2
```

```
*Assign3> (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:

```
*Assign3> system [(Eqn (Prod (RExp 2) (Var "y")) (RExp 10))]
[[2 % 1,10 % 1]]
*Assign3> 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]]
*Assign3> 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]]
*Assign3> 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:

```
*Assign3> (D (Pow (Var "x") (RExp 2)) "x")
D(x^2, x)
*Assign3> (Prod (RExp 5) (D (Pow (Var "x") (RExp 2)) "x"))
5*D(x^2, x)
```

(b) Create a third simplifier based on the second called `simplify3`. 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:

```
*Assign3> simplify3 (D (RExp 2) "x")
0
*Assign3> simplify3 (D (Var "x") "x")
1
*Assign3> simplify3 (D (Var "y") "x")
0
*Assign3> simplify3 (D (Sum (RExp 2) (Var "x")) "x")
1
*Assign3> simplify3 (D (Sum (Var "x") (Var "x")) "x")
2
*Assign3> simplify3 (D (Prod (RExp 5) (Var "x")) "x")
5
*Assign3> simplify3 (D (Prod (Var "x") (Var "x")) "x")
2*x
*Assign3> simplify3 (D (Prod (Var "x") (Sum (Var "x") (RExp 3))) "x")
2*x+3
*Assign3> simplify3 (D (Pow (Var "x") (RExp 2))  "x")
2*x
*Assign3> simplify3 (D (Prod (RExp 5) (Pow (Var "x") (RExp 2)))  "x")
10*x
*Assign3> simplify3 (D (Prod (RExp 5) (Pow (RExp 2) (Var "x")))  "x")
5*D(2^x, x)
*Assign3> simplify3 (D (Prod (Var "x") (Pow (RExp 2) (Var "x")))  "x")
x*D(2^x, x)+2^x
*Assign3> simplify3 (Prod (RExp 5) (D (Pow (Var "x") (RExp 2))  "x"))
10*x
```

Graduate Problems/Undergraduate Extra Credit

1. (10 points) We often differentiate a formula in order to compute a numerical result. There are ways to compute this numerical result directly.

   (a) Recall the definition of the derivative: $\frac{df}{dx}(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$. One way to numerically approximate this result is to let h be a small but finite number. Specifically, let $h = 0.000000001$, and write the function `derivN` that takes a function f and returns a function that approximates f'.

   ```
   *Assign3> (derivN (\x->x^2)) 5
   10.000000082740371
   ```

   (b) It turns out that it is possible to numerically compute the derivative exactly using a technique called *automatic differentiation*. We extend every real number x to the *dual number* $x + y\varepsilon$, where ε is an infinitesimal that satisfies the property $\varepsilon^2 = 0$. A dual number can be represented by a pair $\langle x, y \rangle$.

- Define the type `DualNum` with the one constructor `DualNum` as the type of these pairs. It should have a type parameter so that any `Fractional` type can be used, and it should have a type context to prevent non-`Fractional` types from being used. It should derive `Eq` and `Show`.
- Create instances of `Num` and `Fractional` based on the following rules.
  $\langle u, u' \rangle + \langle v, v' \rangle = \langle u + v, u' + v' \rangle$
  $-\langle u, u' \rangle = \langle -u, -u' \rangle$
  $\langle u, u' \rangle \times \langle v, v' \rangle = \langle u \times v, u \times v' + v \times u' \rangle$
  $1/\langle v, v' \rangle = \langle 1/v, -v'/v^2 \rangle$
  A constant real number c becomes the dual number $\langle c, 0 \rangle$.
- Write a function `d` that takes a function on dual numbers and returns a function on numbers (its derivative). HINT: Given a real number x, apply the function to the dual number $\langle x, 1 \rangle$.
  ```
  *Assign3> (d (\x->x^2)) 5
  10.0
  ```
- Write a function `newtonImprove` that takes a function on dual numbers and returns a function on numbers that improves a guess concerning the zeros of the function using Newton's method. Recall that Newton's method improves the guess x by computing $x - \dfrac{f(x)}{f'(x)}$. HINT: Do *not* use the previous function. Extract both the value of the function and its derivative from the dual number.
  ```
  *Assign3> (newtonImprove (\x->x^2-2)) 1
  1.5
  ```
- Write a function `curtApprox` that takes a dual number and returns a sequence of numbers approximating the cube root of the given number.
  ```
  *Assign3> curtApprox 8
  1.0,3.3333333333333335,2.462222222222222,2.081341247671579,
  2.003137499141287,2.000004911675504,2.0000000000120624,...
  ```
- Write a function `curt` that takes a number and returns its cube root by taking the limit of the sequence of numbers approximating the cube root of the given number.
  ```
  *Assign3> curt 8
  2.0
  ```

2. (30 points) First, add an operators data type (called `Op`) that represents the functions sine, cosine, e^x, and log. They should be called `Sin`, `Cos`, `Exp`, and `Ln` respectively. Then add an operator application choice to the expression data structure called `Ap`. (For example, $\cos(x)$ would be expressed as `Ap Cos (Var "x")`.) Also add the integration operator `Int` choice to the expression data structure. 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:
$$\int x \sin(x^2)\,dx = \frac{1}{2}\int \sin(u)\,du = -\frac{1}{2}\cos(x^2)$$