CS 359: Programming Paradigms
PEX3: Symbolic Differentiation, or
"First Semester Calculus in One Easy PEX"
Due Date: Lesson 30
(70 points)

Help Policy:

AUTHORIZED RESOURCES: Any, except another cadet's program.

NOTE:

- Never copy another person's work and submit it as your own.
- Do not jointly create a program.
- You must document all help received from sources other than your instructor.
- **DFCS will recommend a course grade of F for any cadet who egregiously violates this Help Policy or contributes to a violation by others.**

Documentation Policy:

- You must document all help received from any source other than your instructor.
- The documentation statement must explicitly describe WHAT assistance was provided, WHERE on the assignment the assistance was provided, and WHO provided the assistance.
- If no help was received on this assignment, the documentation statement must state "NONE."
- **Vague documentation statements must be corrected before the assignment will be graded, and may result in a 5% deduction on the assignment.**

1. OBJECTIVES

- Be able to create, test, and debug basic functions in F#
- Develop foundational functions for a symbolic differentiation capability

2. INSTRUCTIONS

In this assignment we will establish some foundational functions for building a symbolic differentiation capability, and then build the capability for some surprisingly powerful functions.

a) The PEX3Template.fsx file on the course web site has header comments for each of the functions that you need to code for this homework assignment[1]. Upon completion, your program will be able to compute derivatives of simple functions and also evaluate for a given value of the independent variable. The differentiation rules that you will first implement are (assume *u* and *v* are functions of x):

$$\frac{dc}{dx} = 0 \quad (\text{c is a constant})$$

$$\frac{dx}{dx} = 1$$

$$\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx}$$

$$\frac{d(uv)}{dx} = u\left(\frac{dv}{dx}\right) + v\left(\frac{du}{dx}\right)$$

$$\frac{d}{dx}\left(u^c\right) = c * u^{c-1} * \frac{du}{dx} \quad (\text{c is a constant})$$

b) Once you've got those working, you will move on to:

$$\frac{d(u-v)}{dx} = \frac{du}{dx} - \frac{dv}{dx} \quad (\textit{difference rule})$$

$$\frac{d\left(\frac{u}{v}\right)}{dx} = \frac{v * \frac{du}{dx} - u * \frac{dv}{dx}}{v^2} \quad (\textit{quotient rule})$$

$$\frac{d(\sin(u))}{dx} = \cos(u) * \frac{du}{dx} \quad (\sin \textit{ rule})$$

$$\frac{d(\cos(u))}{dx} = -\sin(u) * \frac{du}{dx} \quad (\cos \textit{rule})$$

$$\frac{d(\tan(u))}{dx} = \frac{1}{(\cos(u))^2} * \frac{du}{dx} \quad (\tan \textit{rule})$$

$$\frac{d(\ln(u))}{dx} = \frac{1}{u} * \frac{du}{dx} \quad (\textit{natural} \log \textit{ rule})$$

$$\frac{d(\exp(u))}{dx} = \exp(u) * \frac{du}{dx} \quad (\exp \textit{onential rule})$$

[1] This formulation of a symbolic differentiation capability is based upon exercises developed by Hal Abelson and Gerald Jay Sussman at MIT.

The function templates provided in PEX3Template.fsx are intended to walk you through building this symbolic differentiation capability. Code each one in order and then see if some of them can be of use to you in developing the later functions. It is best to code one function and then thoroughly test it before going onto the next function.

c) Your program will accept expressions written as F# differentiated unions. The definition of an expression is at the top of your PEX3 template. For example, your functions would be able to accept:

```
Add(Number 3.0, Variable "x")
```

to mean:

$$3 + x$$

d) Your program must compute the nth derivative of an expression (derivative of the derivatives of the ….) and also evaluate derivatives with a specific value for the independent variable.

e) All of these capabilities must be accessible via the following four functions. All of these functions assume that the input expression expr is an F# discriminated union.

`derivativeInfix expr var`	Computes the derivative of the expression `expr` with respect to variable `var`. Returns a string representing the derivative in infix notation.
`nthDerivativeInfix expr var N`	Computes the Nth derivative of expression `expr` with respect to variable `var`. Returns a string representing the derivative in infix notation.
`evaluateDerivativeInfix expr var value`	Computes the derivative of the expression `expr` with respect to variable `var` and then evaluated at a value of `value` for the variable `var`. Returns a string representing the result in infix notation.
`evaluateNthDerivativeInfix expr var value N`	Computes the Nth derivative of the expression `expr` with respect to variable `var` and then evaluated at a value of `value` for the variable `var`. Returns a string representing the result in infix notation.

f) Your program must be able to return a fully-parenthesized infix representation of the resulting expression. That representation will return a string of the fully-parenthesized expression. For example, you will represent

$$3 + 2x$$

as

```
"(3+(2*x))"
```

3. RESTRICTIONS

This PEX is intended to make you think in a whole new paradigm--the functional paradigm. F# is designed to be a production language. As such, it supports the functional paradigm as well as other paradigms like imperative and even object-oriented. You must complete this PEX using only the functional features of F#. Thus, there are several language constructs you may not use. They are listed below:

1) You may use `let` to define a function, but NOT in an attempt to declare a variable.
2) You may not use `while` or `for`
3) You may not use arrays
4) You may not use the `mutable` keyword
5) You may not use list comprehension (this was allowed in Python, but we don't want to do this in F#)

The PEX can be completed with only the following keywords: `open`, `type`, `of`, `exception`, `if`, `elif`, `else`, `match`, `with`, `raise`, `when`, `let`, `and`, `rec`, and `->`.

4. HELPFUL HINTS

- Please start early! Understanding the functional paradigm thoroughly will make this PEX straightforward. However, it may take some time to gain that understanding.
- The derivative functions in your PEX template are declared with "and" rather than "let". This is the correct syntax due to a subtle typing issue. Don't change those declarations.
- You do not need to implement negation. If you ever need to represent `-x` to mean "negative x", use instead: `Minus(Number 0.0, Variable "x")`
- Carefully note the difference between power and exponential.
  - `Power(Number 5.0, Variable "a")` is equivalent to 5^a
  - `Exponential(Variable "a")` is equivalent to e^a
- Raise intelligent `Pex3Exception`'s at exceptional points in your code. For example, if you don't match a Sine expression in your `derivativeSine` function, do this:
  `raise(Pex3Exception("Unrecognized expression in derivativeSine"))`
  This is not an input into your overall score for the assignment, but it will help debugging.
- Use this website to help you find useful math operators:
  - http://blogs.msdn.com/b/dsyme/archive/2008/09/01/the-f-operators-and-basic-functions.aspx
- To load your functions into the interpreter window for testing, highlight them all (hit Ctrl+A) then hit Alt+Enter.

5. SUBMISSION REQUIREMENTS

a. Your scheme source file (*pex3_<your_last_name>.fs)* must be submitted via Moodle by the beginning of class Lesson 30. Don't forget to include a documentation statement on the Moodle web form.
b. No hardcopy or handwritten submissions are required or acceptable

CS359: PEX3 Cut Sheet

Name:

Grade: **/70**

Implementation			
Make functions are implemented properly (e.g. makeSum, makeDifference, etc…)			**20**
Properly implements replace, evaluate, and toInfixString			**18**
Properly implements derivative functions			**24**
Properly implements the four main functions in table above			**8**
	Implementation Subtotal:		**70**
Miscellaneous			
Quality of program design (white space is not used to improve readability, functions are too large, inappropriate imperative language constructs are used)			**-5**
Comments do not increase clarity of the code or are not used at all			**-5**
	Miscellaneous Subtotal:		**-10**
Adjustments	**Vague/Missing Documentation:**	**–**	**-3.5**
	Late Penalties	**–**	**25/50/75% cap**
	Early Turn In Bonus:	**+**	**3.5**
	Total with Adjustments:		**70**