# Programming Assignment #3
# Functional Programming Haskell

CS-4337.HON – Organization of Programming Languages

**Due: 11:59PM, April 23, 2017**

Define and test the functions described below. In some cases below you may be restricted to which helper functions you may use from the Haskell standard library.

**Your implementation of each function must have its function name spelled *exactly* as it is in the description.**

Your submission should consist of a *single* Haskell source code file. This may be accomplished by defining your functions in a text editor (Notepad, Notepad++, etc). Your Haskell source file should be named using your NetID, beginning with a capital letter. Example: `Cid021000.hs`. The first line of your source code file should be: `module <netid> where`. For example:



```
module Cid021000.hs where
```



Upload this file to eLearning.

In addition to class lectures and eLearning posted examples, you may find Haskell language help here:

- Learn You A Haskell For Great Good
- Haskell (official website)
- Haskell Wiki

## 1. divbyn

Define *your own* Haskell function that takes a single integer as an argument and returns function takes a single integer as an argument and returns a Boolean that indicates whether the number is evenly divisible by its argument.

Your function should return an error (using the built-in error function) if the argument is a non-integer.

**Input**: An integer.

**Output**: A function that accepts and integer and returns a Boolean.

**Example**:

```
Prelude> divbyn 7 14
True

Prelude> divbyn 3 8
False

Prelude> let divby5 = divbyn 5
Prelude> divby5 110
True
```



## 2. f3

Define a Haskell function that takes a function as its argument and passes the number 3 to that function. The function argument must be able to accept a single number value as its argument.

**Input**: A function (named or anonymous) which takes a single number as an argument.

**Output**: The value returned by applying the named function to the number 3.

**Example**:

```
Prelude> f3 sqrt
1.7320508075688772

Prelude> f3 (\ x -> x + 7)
10

Prelude> f3 succ
4
```

## 3. mymap

Define *your own* Haskell function that duplicates the functionality of **`map`** from the standard library. You *may not* use the built-in **`map`** function as a helper function.

**Input**: The input to **`mymap`** is a function that takes a single argument and a homogeneous list of elements of the same data type compatible with the function.

**Output**: A new list the same length as the original list whose elements are function applied to each, respectively.

**Example**:

```
Prelude> mymap sqrt [9,25,81,49]
[3.0,5.0,9.0,7.0]

Prelude> mymap succ [6,4,8,3]
[7,5,9,4]

Prelude> mymap (\ n -> n * n) [5,7]
[25,49]

Prelude> mymap even [2,5,7,12]
[True,False,False,True]
```



## 4. myfilter



Define your own Haskell function that duplicates the functionality of **`filter`** from the standard library. You *may not* use the built-in **`filter`** function as a helper function.



**Input**: The input to **`myfilter`** is a function that takes a single argument and a homogeneous list of elements of the same data type compatible with the function. The function should return a Boolean.

**Output**: A new list of a *subset* of original elements that have returned True in response to the input function.

**Example**:

```
Prelude> myfilter even [1,2,3,4,5]
[2,4]

-- In this example assume that a hypothetical function (lessthan5)
-- exists that returns a Boolean value indicating whether its Num
-- argument is less than 5.
Prelude> myfilter lessthan5 [2,6,3,7,4,8]
[2,3,4]
```

## 5. zipper

Define a Haskell function takes two lists as arguments and returns a single list of 2-tuples. The first tuple should be the both first elements from the respective lists. The second tuple should be the second elements from the respective lists, and so on. If one input list is longer than the other, extra elements of the longer list are ignored. *Your implementation must be recursive*.

**Input**: Two lists of elements of any type. Each of the two lists must be homogenous, but the two lists do not have to be the same type of elements as each other. The two lists do not have to be the same length.

**Output**: A new list whose elements are each 2-tuples. The first tuple is composed of the first elements from two input lists respectively, the second tuple is composed of the second elements form the two input lists respectively, etc. If one list is longer than the other, extra elements of the longer list are ignored.

**Example**:

```
Prelude> zipper [1,2,3,4] ['a','b','c','d']
[(1,'a'),(2,'b'),(3,'c'),(4,'d')]

Prelude> zipper [1,2,3] [4,9,5,7]
[(1,4),(2,9),(3,5)]

Prelude> zipper [5] []
[]
```