Assignment 2
Due: June 8, 2023 at 11:59PM

1 Assignment 2 Specification

1.1 Overview

1.1.1 Goals

1. Practice writing F# functions with functions as parameters
2. Practice using higher order functions like `map`

Patterns are a model of something; in our day-to-day experiences, we tend to categorize things by how closely they match some pattern. For example, we enter a classroom and automatically categorize the furniture based on our pattern of chairs and tables. This ability to recognize patterns and form categories allows us to more rapidly make sense of our world. We enter a new classroom and instantaneously we categorize students and chairs correctly. Seldom do we mistake a chair for a student.

Programming patterns take advantage of this natural ability to recognize patterns in problems. We use patterns in problem-solving in at least two ways: by grouping together distinct cases of a problem, and by abstracting (i.e. reducing) a large number of specific cases to a few general cases. By organizing problems with patterns, our understanding of the problem is generally clearer, the solution simpler and more likely to be correct.

For an example of abstraction, we could define the positive integer successor for each specific integer case by:

```
let successor0 = 1;;
let successor1 = 2;;
let successor2 = 3;;
```

or in general by:

```
let successor n = n+1;;
```

F# further uses our pattern recognition and categorization ability by decomposing a single solution (e.g. function) into separate cases or categories of solutions. The pattern syntax is generally an alternative to `if-then-else` statements, something akin to the switch statement in C or Java. For example, the summation of an integer list of numbers can be defined using patterns, where each pattern solves one specific case of the summation problem. A non-pattern and pattern solution is given below:

```
let rec sum L =
  if L=[]  then 0
  else if L.Tail=[] then L.Head
    else L.Head + sum L.Tail;;

let rec sum L =
  match L with
    | [] -> 0
    | h::[] -> h
    | h::t -> h + sum t;;
```

The non-pattern solution requires that we first read the code to form categories in our mind to understand to the solution. The pattern solution defines and organizes the categories visually, avoiding extra effort and reducing the potential for our misunderstanding the solution.

1.2 Implementation

The skeleton code at `https://classroom.github.com/a/zhtTVm9F` is the start of this week's assignment. Follow the setup and import directions to access the code. Write and test the following functions. Use the provided unit test cases to check your solutions. Feel free to write more tests to better understand the program, and include any helper functions you want. If you do include helper functions (like the ones provided in this writeup) include their definitions before the function that calls them in the source code.

You may change any of the provided function signatures to specify parameters or return types, or to declare a function as recursive, but do not change the function names or types or the number/order of parameters. For this and future F# assignments, you should use no library calls besides `.Head` and `.Tail`. There are some libraries that have helpful features, but you should practice implementing functions from simple language constructs.

1.2.1 dup

Define the function `dup` that takes two parameters: a function, and an input value. It should apply the function twice: first on the input value, and afterwards on the result of the function's first evaluation. See examples below for expected behavior.

```
> let inc (x : int) : int = x + 1;;

> dup inc 5;;
val it : int = 7

> let tail (L : 'a list) = L.Tail;;

> dup tail [1;2;3;4;5];;
val it : int list = [3;4;5]

> let sq (x:int) = x * x;;

> dup sq 5;;
val it : int = 625
```

1.2.2 iterF

An iterated function – written mathematically as $f^n(x)$ – is a function that is applied over and over to an input x. For example, $f^3(x) \equiv f(f(f(x)))$, and $f^0(x) \equiv x$. The `dup` function above is calculating an iterated function for the specific case of $n = 2$. Define the generalized function `iterF` to take n as a parameter instead of assuming it's 2. We expect it to behave the same as `dup` when we call it with $n = 2$. (Interesting application: Images of fractals are generated using iterated functions.)

```
> let inc (x : int) : int = x + 1;;

> iterF inc 5 2;;
val it : int = 7

> iterF inc 5 4;;
val it : int = 9

> let tail (L : 'a list) = L.Tail;;

> iterF tail [1;2;3;4;5] 0;;
val it : int list = [1; 2; 3; 4; 5]

> iterF tail [1;2;3;4;5] 3;;
val it : int list = [4; 5]
```

1.2.3 comp

Define the function `comp` so that `comp f g x` is `g(f (x))` where `f`'s parameter and `x` are the same type, and `g`'s parameter is the return type of `f`. Note that the parameter order swaps around the function names from what you might expect, and that not all of the types are guaranteed to be identical.

```
> let above10 x = x > 10;;

> let inc (x : int) : int = x + 1;;

> above10 14;;
val it : bool = true

> comp inc inc 5;;
val it : int = 7

> comp inc above10 10;;
val it : bool = true
```

1.2.4 sqlist

Define a function `sqlist` to square every element of a given list using `sq` and the `map` functional (both defined below).

```
> sqlist [1;2;3;4];;
val it : int list = [1;4;9;16]
```

Use:

```
> let sq x = x * x;;

> let rec map f L =
    match L with
    | [] -> []
    | h::t -> f h::map f t;;
```

State a runtime complexity estimate (e.g. $O(1), O(n), O(n^2)$, etc.) using the `sqlistComplexity` method.

1.2.5 vecadd

Define a function `vecadd` to add two integer lists using `map2` and `add` (both defined below). Assume that the input vectors are the same length.

```
> vecadd [1;2;3] [4;5;6];;
val it : int list = [5;7;9]
```

Use:

```
> let add x y = x + y;;

> let rec map2 f L1 L2 =
    match (L1,L2) with
      | ([], []) -> []
      | (h1::t1, h2::t2) -> f h1 h2::map2 f t1 t2;;
```

1.2.6 matadd

Define a function `matadd` to add two integer matrices using `vecadd` for vector addition and `map2` for recursion. `matadd` is a one line function. Assume that the input matrices have the same dimensions.

```
> matadd [ [1;2]; [3;4] ] [ [5;6]; [7;8] ];;
val it : int list list = [ [6;8]; [10;12] ]
```

1.2.7 ip

Define a function `ip` to compute the inner product of two lists using `map2` and `foldback`. The inner product is the sum of the products of multiplying matching elements in two lists. In the class notes we defined functions such that `foldback add [1;2;3] 0 = 6` and `map2 mul [1;2;3] [4;5;6] = [4;10;18]`. ip is a one line function. Assume that the two lists have the same length.

```
> ip [1;2;3] [1;2;3];;  // = 1*1 + 2*2 + 3*3
val it : int = 14
```

Use:

```
> let rec foldback f L a =
    match L with
      | [] -> a
      | h::t -> f h (foldback f t a);;
```

1.2.8 fizzbuzz

Define a function `fizzbuzz` to build a list that solves the specific variant of the FizzBuzz problem that follows: generate a `string list` of sequential integers (from 1 to `n`), with some special cases. If the number is divisble by 3, the value should be "fizz". If the number is divisible by 5, the value should be "buzz". If the number is divisible by 3 *and* 5, the value should be "fizzbuzz". Otherwise, the value should be the normal string representation of the integer. Use the `map` function and any other helper functions you need.

```
> fizzbuzz 5;;
val it : string list = ["1"; "2"; "fizz"; "4"; "buzz"]
> fizzbuzz 20;;
val it : string list =
  ["1"; "2"; "fizz"; "4"; "buzz"; "fizz"; "7"; "8"; "fizz"; "buzz"; "11";
   "fizz"; "13"; "14"; "fizzbuzz"; "16"; "17"; "fizz"; "19"; "buzz"]
```

Hint: Like Java, break the program down into smaller pieces and verify that each piece works.

2 Testing

Unit testing is supplied for this assignment. These are not exhaustive tests and you should try running the functions with other valid arguments as well. The tests give a strong indication as to which types are expected for function parameters and return values.

Mac directions:

1. Go to **View→Pads→Unit Tests** and a new window should open
2. Click the 'Run All' button and you should get unit-by-unit feedback on test results.

Windows directions:

1. Go to the **Solution explorer** window.
2. Right-click on the top line in the window: **Solution 'comp3350_a2'** file→**Build Solution**.
3. Go to **Test→Windows→Test Explorer**.
4. The test names should appear in the **Test Explorer** window, and you can run them by clicking **Run All** in the **Test Explorer** window.

3 Grading

Be professional. Make results easy to understand and grade. Include only those parts to be graded. Leave comments where necessary, especially if it aids in grading.

Each of the 8 functions is worth about $\frac{1}{8}$ of the assignment grade.