This problem set will give you further experience with using specifications and expose you to functional language features of the popular multi-paradigm programming language Scala, including currying, higher order functions and Streams. It will also provide you some additional experience in assessing the modifiability/ of code in the context of changed requirements.

If you wish, you can use the assignment to gain further experience with pair programming by teaming up with exactly one other individual in the class to pursue the entire problem set.

Well organized and neat formatting of submissions will be favorably assessed. This includes any programmatic output, recorded data and brief essays.

Please turn a single file for the entire problem set on moodle, with a file name of the form:

PS1_FirstName_LastName_NSID.zip

PROBLEM 1: FUNCTIONAL PROGRAMMING [30 MARKS]

Within this problem you will be using the Scala functional elements featured in class to solve some simple examples. By so doing, you'll implement a form of specifications in a functional fashion.

For this problem, please use the Scala Read-Eval-Print loop that we have been using in class.

Within your program, please import the scala math library, as the following:

import scala.math._

You may wish to make use of several useful facts regarding Scala:

- a one dimensional vector can be created using the expression
  Vector(1, 2, 3)
- we can concatenate a string with another value via the syntax
  "MyPrefix"+a
- The length of a string can be obtained via
  str.length

Within this problem, you will want to draw on the examples shown in class, particularly involving the functors (higher order functions) *map*, *filter*, and *reduce*, with each being applied to a location, and taking a function (closure) as an argument.

Submissions with cleaner abstractions and code will be favourably assessed.

1) Suppose we have an Vector of Double values called *vec*. Write a single expression that adds together (as double) the square roots of all non-negative numbers in the vector. Please note that you can use the *sqrt* function to take the square root of an operation.
2) This subproblem contains three subparts
   a. Write a function that takes in a string and Vector and an integer threshold, and which returns a vector of the same size, except that the string has been concatenated as a prefix to each string in the Vector that exceeds the threshold in length. Demonstrate the correct operation of this function.

   b. Then create a function that "Curries" the function created in the last subpart, such that it first takes a string, and then (separately) the other arguments. Demonstrate the correct operation of this function.
   c. Finally, create a function that "Curries" the function created in subpart a in a different way – namely, so that it first takes the Vector, and then (separately) the other arguments. Demonstrate the correct operation of this function.
   d. Please comment on the above: Under what condition might it be favourable to have a curried value of a function (a staged version of that function) that accepts some arguments earlier, and then other arguments later?

3) Write two alternative functions that take in a string and Vector and an integer threshold, and which returns an Vector containing an ordered set of strings. This Vector that is returned includes all elements (strings) whose length of those strings exceeded the length threshold, except that those strings have had the prefix added to their beginning (i.e., as a prefix). Please note that what is sought is similar in many respects to question 2, but a close reading will reveal that while the function sought in question 2 returns a vector of the same size as that passed in, for this case, the returned vector can include a smaller number of elements than the vector passed in.
4) Here, you will process some data
   a. Using Scala's expression such as *(a to b).toVector*, please generates the integers from 1 to 1000 as a vector called "*sampleTimes*".
   b. Create a function that, given an offset, frequency, and amplitude (all Double values) gives a synthetic "Waveform" function (closure). This Waveform function accepts a time as a parameter, and returns the waveform evaluated at that time. Specifically, for time *t*, it will return the value "*offset* + *amplitude* * *sin*(2 * 3.14159 * *frequency* * *t*)".
   c. Create a waveform vector called "*Waveform*" by applying the Waveform function created in the previous subpart to the *sampleTimes*. For your hand-in, use an offset of 10.0, amplitude of 1.0 and frequency of 0.1, but you'll probably want to test with other values. This operation should use the higher order functions that we have discussed in class.
   d. Define a function "*RMSFromMean*" that computers the "Root Mean Square" distance of the points in the Waveform from the mean of the points in that Waveform. Specifically, given an Vector of values, the function computes the square root of the mean of the square of the deviation of each point x_i from the average of those points Average[x_i]. i.e.

$$\sqrt{\frac{\sum_{j=1}^{n} \left(x_j - Average\left[x_i\right]\right)^2}{n}}$$

   In addition to the built-in higher-order functions *map*, *reduce* and *filter*, functions you may find helpful use following *sin* and *pow* functions from the Scala Math library.
   e. Please apply your code for *RMSFromMean* to *Waveform* (using the suggested values above).

   Please hand in all the code that you have created.

PROBLEM 2: ABSTRACTION AND FLEXIBILITY OF CODE [10 MARKS]

Recall that an important goal of maintaining abstraction barriers in code is the ability to minimize the changes required in the codebase from changing the requirements. Please perform two conceptually simple updates for the code for take home exercise 2 (using the version provided) that supported abstraction of the rule to be used to update the space.

Specifically, please modify the provided code in the following ways.
"Exercise2ExampleSolutionWithCellRuleAbstractionAndDynamicallySelectedRuleV2" as follows:

- Change the space to be "toroidal" -- that is, the space should be shaped like a "donut", in that it "wraps around" in both horizontal and vertical directions. That is, rather than having boundaries, the space is unbounded in both horizontal and vertical directions. If we have countRows rows and countColumns columns, the row just above the 0 row is row countRow-1. Similarly, the row just to the left of column 0 is column countColumns–1. This change will not affect most areas of the code (e.g., the readFile and writeFile functionality will be unchanged), but it will affect some core areas of the code). Please be sure to include the pre-conditions and post-conditions (updating where required). Please note that you can combine this task with the next bullet point, as dealing with them in a combined manner may be simpler.
- Until this point, the code has used one dimensional arrays, explicitly (and manually) performing the computations to map two dimensional coordinates into indices into the arrays. Please now change the code to use explicitly 2 dimensional arrays. Upon performing this, you can also deprecate and remove the function *indexForRowCol*.
- Please update the specifications given (e.g., preconditions, postconditions) to reflect the changes above. Please turn in your updated code.
- Now please perform the above using the code in ConwaysGameOfLifeUglyV1.c Please turn in this updated code.
- Please compare and contrast the amount of work that is required for performing this change in the code from exercise 2 vs. in the original (ugly) codebase.

PROBLEM 3: STREAMS IN SCALA [30 MARKS]

Within this problem, you will explore non-strict computation by creating streams in Scala.

In pursuing this assignment, you may wish to recall that given an existing Stream[T] stream and a value v of type T, one can append an element on to that stream with the following syntax:

v #:: stream

a) Define a function which returns a Stream[Long] whose n^{th} element (starting from 1) represents n! (i.e., n*(n-1)*(n-2)*...*1). Please note that you can decide what parameters this function takes, as long as it can return this specified stream.

b) Please create a function (again parameterized in a manner of your choosing) that can return a Stream[Long], where each element of that returned stream is represents each successively larger prime number (represented as a Long). (As with part a, please note that such a stream would typically require that the function be called with the appropriate arguments).

PROBLEM 4 [*FOR GRADUATE STUDENTS ONLY*]: AN INTEGRATION OPERATOR [15 MARKS]

Within this problem, you will implement a higher-order function in Scala. The function will each accept a function as an argument (amongst other things) and – further – returns a function.

In pursuing this home exercise, as in the last one, you may wish to make use of several useful facts regarding Scala:

- recall that one can define a method that is curried ("staged") to accept successive arguments using syntax showing successive parameter lists. An example would be the following:
  def curriedFn(argA: Int, argB: Double)(argC:String) = *Some expression here*
- one can write a method taking a unary function as an argument using the syntax such as the following:
  def higherOrderFunction(fn: Double => Double) = *Some expression here*
- one can define an anonymous function (closure) using the syntax
  args => body
  where args can be a single parameter (e.g., x: Double) or a tuple of multiple parameters (e.g., (x: Double, y: Double))
- Given values *a*, *b*, and *s* that are Double values, one way that you can generate a vector of Double values from *a* to *b* with a step of s is by calling (*a* to *b* by *s*).toVector an example would be (0.0 to 1.0 by 0.1).toVector
- You can import elements from the Scala math libraries such as *exp* via the following import statement: import scala.math._

The function that we will implement here is the integration operator of Calculus. Specifically, given a unary function *f(x)*, we will return a function that can evaluate a numerical approximation to the integral of that function over a user-specified interval [x_{start}, x_{end}]. You may remember that the value of the integral of a function over an interval is the total area under the curve that function if considered only for that interval. As in the previous problem, we will create a numerical approximation to this value.

1) Create a numerical *integration operator*. This operator (implemented as a Scala method, using def) should first take in an Double returning unary function *fnUnary* (which serves as f(x)). This should return a function that takes (in turn) a Double precision value representing the integration step size *dx*, and returns a function which takes (in turn) a starting point *xStart* and ending point *xEnd*, each represented Double precision values, and which then (finally) returns the result of integrating *fnUnary* from *xStart* to *xEnd* according to the "rectangle method" of integration with step size *dx*. (Please note that in the likely event that (*xStart-xEnd*) is not a perfect multiple of *dx*, you may simply take portion of the area of the final "rectangle" that falls within the interval [*xStart*, *xEnd*]). Thus, the computation will consist in phases, where one gets back a function representing the integral, and can then (separately) successively request the value of that integral for various intervals.

The Euler integration scheme that we will use here is particularly simplistic: For each interval of length *dx* over the interval [*xStart*, *xEnd*], we will approximate the value of the function to be integrated (*fnUnary*) as its value of the function. Thus, we can think of the area under the curve *fnUnary* as being approximated by a series of rectangles, each (with the exception of the last) being of width *dx*, and of a height given by the value of *fnUnary* at the value of *x* of the left side of that rectangle. (Thus, we are not using the value of fnUnary at the mid-point, but instead on the left of that rectangle). Please see the following reference for an animated illustration of the

method (bearing in mind that the midpoint approximation also discussed in that page is not required):
https://en.wikipedia.org/wiki/Rectangle_method

The currying requested above can be implemented by returning explicit functions, or by Scala's "curried" successive parameter lists, and using the "_" operator (as shown in class) to treat the results of the partially applied curried function as a function.

2) Using a value of *dx* of 1E-3 for all, please create named functions (declared using Scala's with *val* declaration) representing integrals of each the following functions

f(x)=1
f(x)=x
f(x)=2*x
f(x)=x*x

3) Please test out your integral operator on the functions above by evaluating them on the [0, 2] interval, reporting the results. Do these results look reasonable? What limitations (if any) do you see?

4) Using the derivative operator created in the previous exercise, how could you work to build confidence as to whether your integral operator is working?