# CMPT 383: Vitamin #1

Anders Miltner
miltner@cs.sfu.ca

Due Sep 20

## Introduction

This Vitamin is here to get you accustomed to programming in Haskell. In this Vitamin, you will practice basic coding Haskell control flow, practice list processing in Haskell, and practice simple recursion.

This submission will be autograded. There are some portions of the assignment that are ungraded, and some that will be graded. We provide a (partial) test suite for partial validation. You can run these tests by opening a terminal in the v1 directory, and running `stack test`.

## 1 List-based GCD Algorithm

In this section, you will write a list-based algorithm for finding the greatest common divisor of two numbers. This will provide a place to practice recursion, manipulating integral numbers, list processing, and fleshing out a test suite. In this problem, you will only be handed natural numbers. In other words, you don't need to worry about being provided inputs of 0 or negative numbers.

Open the file "Gcds.hs". In this file, you should see a number of stubbed-out functions – `isDivisor`, `allDivisors`, `listIntersection` and `listGcd`.

The "Gcds.hs" file contains detailed instructions on how to fill out these functions. Follow the instructions provided in that file.

## 2 Purely Functional Queue

A queue is a fundamental data structure, and is critical for infrastructure like packet processing, CPU scheduling, network routers, and more.

All of these applications require incredibly efficient processing. Having O(1) enqueing and dequeing is critical in such situations.

Typically, these queues are encoded with a doubly-linked list. When an element is enqueued, it is added to the front end of the list, and when an element is dequeued, it is removed from the back end of the list. This is possible in O(1) because references are maintained to the front and back ends of the list.

However, references and doubly-linked lists are both impossible in functional languages like Haskell. So we must use a different approach. In Haskell, rather than using a doubly-linked list, we will use two interacting lists. By doing this, we can achieve purely functional queues, while maintaining (amortized) O(1) enqueues and deques.

To achieve this, you will write three functions: Peek, Pop, and Push. Peek will return the next element to be dequeued. Pop will remove the next element to be dequeued. Push will enqueue an element. Furthermore, you will need to encode the empty list.

Intuitively, the way this queue is implemented is that there is a "popping" list and a "pushing" list. When elements from the "popping" list are exhausted, the "pushing" list is reversed, and becomes the "popping" list, and the pushing list is set to empty.

Open the file "Queue.hs". In this file, you should see a number of stubbed-out functions – `empty`, `enqueue`, and `dequeue`. The "Queue.hs" file contains detailed instructions on how to fill out these functions. Follow the instructions provided in that file.