# PLC: Workout 4 [90 points]

Due date: Wednesday, February 19th by midnight

## About This Homework

This assignment is about input/output (IO) in Haskell, and the functor, applicative, monad abstractions.

### How to Turn In Your Solution

Please submit your solution via ICON. The required file for this assignment is:

- `Main.hs`
- `Exercises.hs`
- `Translations.hs`

**Please use exactly the file names we are requesting.** We will require you to resubmit your homework with a 5-point penalty if the names are not exactly as we are requesting. This is for purposes of grading scripts. It is ok if ICON adds a number to your file name on multiple submission (which is allowed up to the deadline).

### Partners Allowed

You may work alone or with one partner. You should both turn in your solution to the assignment, which we expect will be the same (but is allowed to be different, if you worked together but then you decided to add to your solution – or whatever the scenario). Also, you need to turn in a file called `partner.txt` which lists your partner's Hawkid. This will let us know that you worked with that person (lest we incorrectly think you plagiarized another student's similar submission).

### How To Get Help

You can post questions in the `workouts` section on Piazza.

You are also welcome to come to our office hours. See the course's Google Calendar, linked from the github page for the class, for the locations and times for office hours.

## 1 Reading

Read Chapter 12 of *Programming in Haskell*.

## 2 Main.hs [45 points]

Solve the problems `p1` through `p5` in `Main.hs`. These all construct IO actions for some basic examples of input/output. You can run the `main` function in that file to test your solutions. [9 points each]

## 3 Exercises.hs

This file has several parts.

### 3.1 Working abstractly with `Applicative` and `Monad` [12 points]

Solve problems `a1` and `a2`. There are test cases for these in `PublicTests.hs`. [6 points each]

### 3.2 Translating `do`-notation to bind [12 points]

Functions `s1` and `s2` use `do`-notation to carry out some monadic operations. Write functions `t1` and `t2` which do the same operations but use the basic monadic bind operator (`>>=`) instead. (So `t1` and `t2` should not use `do`.) Please put your solutions in a new file called `Translations.hs`. [6 points each]

### 3.3 Implementing a stack machine [21 points]

The last part of `Exercises.hs` defines a `St` monad for maintaining a stack of `Int`s. There are basic operations `pushValue`, `topValue`, and `popValue` to access and modify that stack. Also defined is a datatype `StackProg` representing simple programs that operate on the stack:

- `Push x` pushes `Int x` onto the stack
- `Pop` pops the top element off the stack
- `BinOp f` pops the top two elements `x` (topmost) and `y` (next after `x`) off the stack, applies the function f to them in order (so `f x y`) and pushes that result back onto the stack
- `Flip` flips the top two elements on the stack
- `p1 :- p2` executes the `StackProg p1` first, and then the `StackProg p2`.
- Implement the `sum3` function which will take three `Int`s and create a `StackProg` that sums them up, leaving the sum as the sole value at the top of the stack. [6 points]
- Implement `evalh` which takes a `StackProg` and evaluates it, making use of the operations on the `St` monad to manipulate the stack. [15 points]