# COMP3400
# 2024
# Assignment 2 Written

Paul Vrbik

March 28, 2024

In addition to this written work there are *four* coding questions. The written work is worth 40 points and the coding questions are worth 40 points totalling 80 points.

## Tail Recursion

The *mean* of a collection of observations $x_1, x_2, \ldots, x_n$ is given by

$$\bar{x}_n = \frac{1}{n} \sum_{k=1}^{n} x_k.$$

for $n \in \mathbb{N}$, $n > 0$. That is to say, the above is an equation that computes the mean using *all* the values $(x_1, x_2, \ldots, x_n)$.

**Question 1.** *Medium* [1 MARK]

Produce an equation which computes $\bar{x}_{n+1}$ from *only* $(n, \bar{x}_n, x_{n+1})$ for $n \in \mathbb{N}$, $n > 0$.

**Question 2.** *Medium* [2 MARKS]

Define a *linear recursive*

```
lrMean :: [Float] -> Float
```

that computes the mean of a list. Your function must use `lrMean xs` to compute `lrMean (x:xs)` and have a patten defined for empty input. Note, `fromIntegral.length` is compatible with `Float`.

**Question 3.** *Easy* [1 MARK]

Briefly (no more than two sentences) justify your definition of `lrMean []`.

**Question 4.** *Medium* [4 MARKS]

Define a *tail recursive* helper function with type:

```
trMean :: Float -> Float -> [Float] -> Float
```

that finds the mean of *non-empty* list.

Remember your function may *only* be equal to

1. a call to itself with different inputs, or
2. one of the inputs.

In particular your base case *cannot* do any more function calls *including* any arithmetic.

**Question 5.** *Easy* [1 MARK]

Define

```
mean :: [Float] -> Float
```

via a single call to `trMean`.

**Question 6.** *Easy* [1 MARK]

Define an iteration invariant for `trMean` that proves the correctness of `mean` for nonempty input.

**Question 7.** *Medium* [5 MARKS]

Prove `trMean` satisfies your iteration invariant for nonempty input.

**Question 8.** *Easy* [1 MARK]

State the bound value for `trMean`.

**Question 9.** *Easy* [2 MARKS]

Prove your bound value is always non-negative and decreasing.

**Question 10.** *Medium* [4 MARKS]

Define *four* distinct quick-checks for `mean` that *all* use lists from `Arbitrary [Float]`.

In particular, your quick-checks should be for lists of *arbitrary length* and genuinely check *properties*.

# Induction

**Question 11.** *Medium* [8 MARKS]

Consider the following definitions for implementing addition on natural numbers.

```
1 data Nat = Zero | Succ Nat deriving Show

2 plus :: Nat -> Nat -> Nat
3 plus m Zero     = m
4 plus m (Succ n) = plus (Succ m) n
```

Prove that `plus` is *associative*. That is, prove:

```
plus (plus m n) k = plus m (plus n k)
```

You may *take for granted* that

```
5 plus m (Succ n) = plus (Succ m) n = Succ (plus m n)
```

When justifying your steps use the line numbers on this page.

*Hint:* Do induction over `k` while letting `m` and `n` be free.

# Expression Functor

**Question 12.** *Medium* [7 marks]

Consider the following datatype for encoding expressions that adds values.

```
1 data Expr a = Const a | Add (Expr a) (Expr a) deriving Show
```

with the following functor definition…

```
2 instance Functor Expr where
3     -- fmap :: (a->b) -> Expr a -> Exp b
4     fmap f (Const a) = Const $ f a
5     fmap f (Add x y) = Add (f <$> x) (f <$> y)
```

Prove that your functor instance for `Expr a` satisfies the *second* functor law:

```
6 fmap (g . h) = fmap g . fmap h
```

When justifying your steps use the line numbers on this page.

# Applicatives

**Question 13.** *Hard* [3 MARKS]

The dual application operator `(<**>)` from `Control.Applicative` changes the direction of calculations but not the effects:

```
infixl 4 <**>
(<**>) :: Applicative f => f a -> f (a -> b) -> f b
(<**>) = liftA2 (flip ($))
```

We define another operator `(<*?>)` with the same type as `(<**>)`, but a different implementation:

```
infixl 4 <*?>
(<*?>) :: Applicative f => f a -> f (a -> b) -> f b
(<*?>) = flip (<*>)
```

Provide an example of two Applicative calculations `a1` and `a2` for which `a1 <**> a2` is *not* the same as `a1 <*?> a2`.

Link to documentation for `liftA2`

Link to documentation for `flip`