# CS355: Programming Paradigms Lab

## Lab 2: Higher Order Functions

August $12^{th}$, 2024

---

**Q1.** Once more recall Newton's procedure to compute square roots. Write a procedure `iterative-improve` that takes two procedures as arguments: a procedure for telling whether a guess is good enough and another for improving a guess. The procedure `iterative-improve` should return as its value another procedure that takes a guess as argument and keeps improving the guess until it is good enough. Rewrite the `square-root` procedure from class in terms of `iterative-improve`.

**Q2.** Recall the `sum-series` procedure from class that, given a way to compute a `term` and a way to compute the `next` term, computes the sum of integers in the range `a` to `b`.

(A) Write an analogous procedure called `product` that returns the product of the values of a function at integers over a given range.

(B) Define a `factorial` procedure in terms of `product`.

(C) If your `product` procedure generates a recursive process, write one that generates an iterative process. If it generates an iterative process, write one that generates a recursive process.

**Q3.** Let `f` and `g` be two one-argument functions. The *composition* `f` after `g` over an input `x` is defined to be the functions `f(g(x))`. Define a procedure `compose` that implements composition. For example, if `inc` is a procedure that adds `1` to its argument, `((compose square inc) 6)` should return 49.

**Q4.** Recall the functional programmer's claim that everything apart from lambdas is syntactic sugar. Let us get a flavour of the same.

Say we define the first number `zero` as:

```
(define zero (lambda () '()))
```

where, for now, say `'()` represents *empty*.

Say we additionally have the following two definitions for computing the successor and the predecessor:

```
(define (succ x) (lambda () x))
(define (pred x) (x))
```

The purpose of the above two clever definitions is to ensure the property that the predecessor of the successor of a number $n$ is $n$. Make sure you see this in the definitions (in terms of when is a procedure applied versus defined, etc.).

What is the number one? Successor of zero! What is the number two? Successor of one (or successor of successor of zero)! And so on. Let's define a few numbers this way:

```
(define one (succ zero))
(define two (succ one))
(define three (succ two))
(define four (succ three))
(define five (succ four))
```

Now say we have a way to compute if something is zero:

```
(define (is-zero? x) (null? (x)))
```

where `null?` is a predefined procedure in Scheme.

Next let us define a procedure that checks if two of our newly defined *Church numerals* are equal:

```
(define (is-equal? x y)
  (cond ((is-zero? x) (is-zero? y))
    ((is-zero? y) (is-zero? x))
    (else (is-equal? (pred x) (pred y)))))
```

Basically, as the only actual equality check we have is for zeros (in the form of `is-zero?`), what we are doing in `(is-equal? x y)` is getting down to using `is-zero?` by recursively reducing `x` and `y` using the `pred` function.

(A) Write code to verify the following kinds of properties:

1. `zero` is not equal to one, but is equal to `zero`
2. `four` is equal to `succ(succ(succ(succ(zero))))`
3. The predecessor of the successor of `two` is `two`

Apart from checking for equality, what else are numbers useful for? Addition, subtraction, multiplication, and so on. Understand and programmatically verify (like the items above, using the `is-equal?` function) that the following `add-church` procedure works:

```
(define (add-church x y)
  (if (is-zero? y)
    x
    (add-church (succ x) (pred y))))
```

(B) Now write your own versions of `subtract-church` and `multiply-church`.

Absorb the fact that we were able to implement *numbers* and (some of) the important associated operations just with *lambdas* (recall the representation of our 'zero'); that should give you a flavour of the expressive power of the **lambda calculus**!