# B490 HOMEWORK 1: EQUATIONAL REASONING

This homework is due on Tuesday, January 19, 2016 by 11:15am. Submit online a plain text file titled `username_hw1.txt`, containing your *name*, your *username*, and your solutions to each of the exercises. Here's the grading policy for late homework: if you submit it by the next lecture, you get half credit; otherwise, you get no credit.

## 1. Beta-reductions and eta-reductions (50%)

Rewrite each expression below to its beta/eta-normal form. Show your work as a sequence of equations, each a single beta-reduction or eta-reduction.

(1) `(λx. λy. x-y) 3`

(2) `((lambda (f) (lambda (x y) (f y x))) (lambda (x y) (- x y)))`

(3) `(λf. λx. λy. (f y) x) (λx. λy. x-y) 3`

(4) `((lambda (x) (lambda (y) (- x y))) y)`

(5) `(λfx. f (f x)) (λfxy. f y x)`

(6) `(λfx. f (f x)) (λfxy. f y x) (λx. λy. x-y) 3`

(7) `(λfx. f (f x)) ((λfxy. f y x) (λx. λy. x-y) 3)`

(8) `((lambda (f) (f f)) y)`

(9)
```
((lambda (sqr) (lambda (x y) (sqrt (+ (sqr x) (sqr y)))))
 (lambda (x) (* x x)))
```

(10)
```
(let ((sqr (lambda (x) (* x x))))
  (lambda (x y) (sqrt (+ (sqr x) (sqr y)))))
```

## 2. Equivalence (30%)

For each pair of terms below, are they equivalent? If yes, prove it by showing a sequence of equations. If no, prove it by showing a context that tells them apart.

(11) the terms `((λx. λy. x-y) 3)` and `(λy. y-3)`

(12) the terms `((λx. λy. x-y) 3)` and `(λy. 3-y)`

(13) the terms `(lambda (x) (lambda (y) x))` and `(lambda (x) (lambda (x) x))`

(14) the terms `(λx. x-y)` and `(λx. x-z)`

(15) the terms `(lambda (x) ((f 3) x))` and `(f 3)`

(16) the terms `(λx. λy. x+y)` and `((λf. λx. λy. (f y) x) (λx. λy. x+y))`

## 3. Program calculation (20%)

Consider the following definitions of the list functions `append` and `reverse` in Scheme:

```
(define append
  (lambda (xs ys)
    (cond
      ((null? xs) ys)
      (else (cons (car xs) (append (cdr xs) ys))))))

(define reverse
  (lambda (xs)
    (cond
      ((null? xs) '())
      (else (append (reverse (cdr xs)) (list (car xs)))))))
```

On a long list, `reverse` takes a long time. We can use equational reasoning to calculate a faster implementation of `reverse`. First, let's create a new function that combines `append` and `reverse`:

```
(define append-reverse
  (lambda (xs ys)
    (append (reverse xs) ys)))
```

For each of the three calculations below, show your work as a sequence of equations, and justify each equation by a few words.

(17) Calculate `(append-reverse '() ys)` to get a result that uses neither `append` nor `reverse`.

(18) Calculate `(append-reverse (cons x xs) ys)` to get a result that uses neither `append` nor `reverse`.

(19) Combine the two results above into a new implementation of `append-reverse` that uses neither `append` nor `reverse` but rather calls itself recursively.

(20) Calculate `(reverse xs)` to get a result that uses neither `append` nor `reverse` but rather calls `append-reverse`. Finally, turn this result into a new and faster implementation of `reverse`.