YOUR NAME:
_____

Instructions:

1. Please make sure you are sitting at the seat assigned to you.

2. Please write your name on both the exam booklet and the answer sheet.

3. You MUST enter your answers into the answer sheet. Each problem that requires an answer has been numbered. Place your answer at the corresponding number on the answer sheet. Please use the empty space on the exam booklet to show your work.

4. While there may be a lot of reading to do on a problem, there is relatively little code to write, so please take the time to read each problem carefully.

5. Note that any procedures or code fragments that you write will be judged not only on correct function, but also on clarity and good programming practice.

6. Talking, passing around pencils and erasers etc. are not allowed. Please raise your hand if you have a question and a TA will be with you. No questions during the last 30 minutes.

7. You have 120 minutes. Good luck!

[Part 1. Short Questions (15 points)]

Question 1.1: Rewrite the following code fragment to use cond:

```
(if (< x -2)
    (* x -2)
    (if (< x 2)
        (* x x)
        (* x 2)))

;;; (cond ((< x -2) (* x -2))
;;;       ((< x 2) (* x x))
;;;       (else (* x 2)))
```

Indicate the type of each of the following procedures. Use the symbols "->" to denote "maps to", for example, the procedure square has type "number->number". Use the following terms to describe primitive types of data: number, boolean, string.

Question 1.2:

```
(define (test bar n) (if (bar n) n (test bar (+ n 2))))

;;; (number -> boolean), number  -> number
```

Question 1.3:

```
(define (test foo bar n) (if (bar n) #t (test foo bar (+ n (foo n)))))

;;; (number -> number), (number -> boolean), number  ->  boolean
```

Question 1.4:

```
(define (test foo bar n) (if (bar n) (+ 1 (foo n)) (test foo bar (+ n 3))))

;;; (number -> number), (number -> boolean), number  ->  number
```

Indicate the value of the following expressions. If the value is a procedure write "procedure" and indicate its type. If the expression results in an error, write "error" and indicate its reason.

Question 1.5:

```
(((lambda (+) (lambda (* /) (+ * /)))
  -)
 2 3)

;;; Value: -1
```

Question 1.6:

```
(define f (lambda (*) (lambda (a b) (a * b))))
((f 3) - 5)

;;; Value: -2
```

Question 1.7:

```
((lambda (a b) (a b))
 (lambda (c) (* 2 c))
 10)

;;; Value: 20
```

[Part 2. Three Types of Multiplication (15 points)]

Suppose we want to implement multiplication, but only have the operations of addition, subtraction, and simple predicates available to us. In this question, you will implement three versions of multiplication. Each version will take two arguments, a and b, both of which you may assume are positive integers, and return the product of a and b, using only these operations.

Question 2.1. Implement a recursive procedure for multiplication.

```
;;; (define (mul a b)
;;;   (if (= a 1) b
;;;       (+ b (mul (- a 1) b))))
```

Question 2.2. Implement an iterative procedure for multiplication.

```
;;; (define (mul a b)
;;;   (define (helper i result)
;;;     (if (= i 0) result
;;;         (helper (- i 1) (+ result b))))
;;;   (helper a 0))
```

Question 2.3. Implement a logarithmic time procedure for multiplication using the identity:

$$a * b = (a/2) * (2b)$$

You can use the operations "double", "halve" and "even?". Assume that "halve" will behave correctly only if its argument is even.

```
;;; (define (mul a b)
;;;   (cond ((= a 1) b)
;;;     ((even? a) (mul (halve a) (double b)))
;;;     (else (+ b (mul (- a 1) b)))))
```

[Part 3. Recursive and Iterative (15 points)]

We want to implement a procedure to compute the function f(n) for positive integers n defined as:

f(n) = f(n−1) + 2 f(n−2) + 3 f(n−3) for n > 3
f(n) = n   for n <= 3

Question 3.1. Write a recursive procedure to implement f(n).

```
;;; (define (f n)
;;;   (if (<= n 3) n
;;;       (+ (f (- n 1))
;;;          (* 2 (f (- n 2)))
;;;          (* 3 (f (- n 3))))))
```

Question 3.2. Write an iterative procedure to implement f(n).

```
;;; (define (f n)
;;;   (define (helper k f1 f2 f3)
;;;     (if (= n k) f3
;;;         (helper (+ 1 k) f2 f3 (+ f3 (* 2 f2) (* 3 f1)))))
;;;   (if (<= n 3) n
;;;       (helper 3 1 2 3)))
```

Question 3.3. What is the running time for your two implementations?

```
;;; Recursive: exponential running time
;;; Iterative: linear running time
```

[Part 4. List procedures (20 points)]

Question 4.1 (5 points). Write a procedure (max-list lst) that takes a list of positive integers (possibly containing duplicates) and returns the largest element of that list. You can use the operations < and > to compare the elements of the list. If the argument lst is the empty list, your procedure should return 0.

```
;;; (define (max-list lst)
;;;   (if (null? lst) 0
;;;       (let ((a (car lst))
;;;             (b (max-list (cdr lst))))
;;;         (if (> a b) a b))))
```

Question 4.2 (5 points). Write a procedure (min-list lst) that takes a list of positive integers (possibly containing duplicates) and returns the smallest element of that list. You can use the operations < and > to compare the elements of the list. If the argument lst is the empty list, your procedure should return the value of the variable *infinity*.

```
;;; (define (min-list lst)
;;;   (if (null? lst) *infinity*
;;;       (let ((a (car lst))
;;;             (b (min-list (cdr lst))))
;;;         (if (< a b) a b))))
```

Question 4.3 (10 points). Identify the common parts and the differences between the two procedures you have implemented. Implement a higher order procedure best-list that returns the best element of a list according to a given criterion. Best-list should use the common structure of min-list and max-list, and take the variable parts as arguments. Show how you can implement min-list and max-list using best-list.

```
;;; (define (best-list lst cmp default)
;;;   (if (null? lst) default
;;;       (let ((a (car lst))
;;;             (b (best-list (cdr lst) cmp default)))
;;;         (if (cmp a b) a b))))
;;;
;;; (define (max-list lst) (best-list lst > 0))
;;; (define (min-list lst) (best-list lst < *infinity*))
```

[Part 5. Ackermann's function (20 points)]

The following procedure computes a mathematical function called Ackermann's function.

```
(define (A x y)
  (cond ((= y 0) 0)
        ((= x 0) (* 2 y))
        ((= y 1) 2)
        (else (A (- x 1)
                 (A x (- y 1))))))
```

Compute the value returned by the following expressions:

Question 5.1 (3 points). (A 1 3)
;;; 8

Question 5.2 (3 points). (A 2 3)
;;; 16

Question 5.3 (3 points). (A 3 3)
;;; 65536

Write down the mathematical expression computed by the following procedures. For example given (define (f n) (+ (* 2 n) 1)), you would write f(n) = 2n+1.

Question 5.4 (3 points). (define (f n) (A 0 n))
;;; f(n) = 2n

Question 5.5 (4 points). (define (g n) (A 1 n))
;;; g(n) = 2^n

Question 5.6 (4 points). (define (h n) (A 2 n))
;;; h(n) = 2^(2^(2^...(2))) with n two's.

[Part 6. Higher order list procedures (15 points)]

We have seen three higher order list procedures in class. Here are some examples of their use:

```
(filter odd? (list 1 2 3 4)) => (1 3)

(map square (list 1 2 3 4)) => (1 4 9 16)

(accumulate + 0 (list 1 2 3 4)) => 10
```

Question 6.1. Implement a procedure (size vec) that will calculate the size of an n−dimensional vector whose coordinates are given in the list vec. Use map, filter, and/or accumulate in your solution. Remember that the size of a vector is the square root of the sum of the squares of the coordinates. Do not use cons, car, or cdr.

```
;;; (define (size vec)
;;;   (sqrt (accumulate + 0 (map square vec))))
```

Question 6.2. Implement a procedure (remove num lst) that removes all occurances of a given number "num" from a list of numbers "lst". Use map, filter, and/or accumulate in your solution. Do not use cons, car, or cdr.

```
;;; (define (remove num lst)
;;;   (filter (lambda (x) (not (= x num))) lst))
```

Question 6.3. Reimplement the procedure max−list from Question 4.1 using accumulate. Do not use cons, car, or cdr. You may use the two argument procedure "max".

```
;;; (define (max−list lst)
;;;   (accumulate max 0 lst))
```

Here are the definitions of the three higher order list procedures for your reference:

```
(define (filter pred lst)
  (cond ((null? lst) nil)
        ((pred (car lst))
         (cons (car lst) (filter pred (cdr lst))))
        (else (filter pred (cdr lst)))))

(define (map proc lst)
  (if (null? lst) nil
      (cons (proc (car lst))
            (map proc (cdr lst)))))

(define (accumulate op init lst)
  (if (null? lst) init
      (op (car lst)
          (accumulate op init (cdr lst)))))
```