Final Exam Information
Here are the instructions for the Winter 2020 Final Exam. Please review before the exam day.

There are 6 questions.
1. The first question consists of 10 True-or-False questions.
2. One question will ask you about the environment model of evaluation and scope (lexical and dynamic).
3. One question will ask you about basic knowledge of interpreters, both calc and racket-1. The source code for these interpreters is provided on culearn and at the end of this document.
4. There are 3 questions that require Racket coding. The questions will ask you to write a piece of code to solve a problem. You answer will be copied and pasted into DrRacket and graded according to the output expected.

<u>Programming questions</u>

i. One question will ask you to write a procedure that takes a list and and operates on it, similarly to the examples below:
   - Write a procedure that accepts an input list P and returns an output list that
     a. filtered out negative numbers
     b. filtered out negative numbers and sorted the list from min to max
     c. filtered out negative numbers, sorted the list from min to max and added an extra item at the end of the list.
   - Write a procedure gamma that accepts two input lists and returns an output list that
     a. combines two list together by summing each item i.e. `(gamma '(1 2 3) '(4 5 6))` returns '(5 7 9)
     b. squares each item in both lists before summing them i.e. `(gamma '(1 2 3) '(4 5 6)) returns '(17 29 45)`
     c. adds a summation of all items in the front of the list i.e. `(gamma '(1 2 3) '(4 5 6)) returns '(91 17 29 45)`

   Note: you can use any list manipulating procedures available from Racket e.g. append, sort, foldl, foldr, map, etc.

   Your procedure will be graded accordingly– For example: a -> 1 marks, b -> 3 marks, c -> 5 marks.

ii. One question will ask you to solve a problem using tree recursion. You may have to write a recursive procedure, a higher-order procedure, or write procedures that call the higher-order procedure. SICP section 1.2.2 , Tree recursion link on culearn.

iii. One question will ask you to write a procedure using mutation. See Mutation! Sildes and SICP section 3.1.1, 3.1.3 Exercise 10.1, 10.2 in Evans

In most cases, you code will be graded using DrRacket. However, it's always a good idea to make your intentions clearer by commenting your code MEANINGFULLY. Please don't write comments that state the obvious; for example, there is no need for comments like this one:

```
(+ x 1) ; Add 1 to x
```

Write only the comments to clarify anything you think requires further explanation; for example, to summarize what short chunks of code do, especially if you're not sure if your code is correct. As an example:

```
                    ; Sum all integers in the list that are divisible by 5.
```

Recommended exercises from Introduction to Computing (URL: http://computingbook.org/).
Chapter 3: all 11 exercises (at this point in the course, these should be very easy).
Chapter 4: 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9, 4.10, 4.13, 4.14.
Chapter 5: all exercises except for 5.24 and 5.25.
Chapter 9: 9.1, 9.2, 9.3.
Chapter 10: 10.1, 10.2
SICP https://web.mit.edu/alexmv/6.037/sicp.pdf
SICP 3.1 3.2 3.10 3.11

calc interpreter

```
;; Racket calculator -- evaluate simple expressions

; The read-eval-print loop:

(define (calc)
  (display "calc: ")
  (flush-output)
  (print (calc-eval (read)))
  (newline)
  (calc))

; Evaluate an expression:
(define (calc-eval exp)
  (cond ((number? exp) exp)
        ((list? exp) (calc-apply (car exp) (map calc-eval (cdr exp))))
        (else (error "Calc: bad expression: " exp))))
(trace calc-eval)

; Apply a function to arguments:
(define (calc-apply fn args)
  (cond ((eq? fn '+) (foldr + 0 args))
        ((eq? fn '-) (cond ((null? args) (error "Calc: no args to -"))
                           ((= (length args) 1) (- (car args)))
                           (else (- (car args) (foldr + 0 (cdr args))))))
        ((eq? fn '*) (foldr * 1 args))
        ((eq? fn '/) (cond ((null? args) (error "Calc: no args to /"))
                           ((= (length args) 1) (/ (car args)))
                           (else (/ (car args) (foldr * 1 (cdr args))))))
        (else (error "Calc: bad operator:" fn))))
```

racket-1 interpreter

```
. #lang racket

;; The "read-eval-print loop" (REPL):

(define (racket-1)
  (newline)
  (display "Racket-1: ")
  (flush-output)
  (print (eval-1 (read)))
  (newline)
  (racket-1)
  )

(define (eval-1 exp)
  (cond ((constant? exp) exp)
    ((symbol? exp) (eval exp))  ; use underlying Racket's EVAL
    ((quote-exp? exp) (cadr exp))
    ((if-exp? exp)
     (if (eval-1 (cadr exp))
         (eval-1 (caddr exp))
         (eval-1 (cadddr exp))))
    ((lambda-exp? exp) exp)
    ((pair? exp) (apply-1 (eval-1 (car exp))      ; eval the operator
               (map eval-1 (cdr exp))))
    (else (error "bad expr: " exp))))

(define (apply-1 proc args)
  (cond ((procedure? proc)  ; use underlying Racket's APPLY
     (apply proc args))
    ((lambda-exp? proc)
     (eval-1 (substitute (caddr proc)   ; the body
              (cadr proc)    ; the formal parameters
              args           ; the actual arguments
              '())))     ; bound-vars, see below
    (else (error "bad proc: " proc))))


(define (constant? exp)
  (or (number? exp) (boolean? exp) (string? exp) (procedure? exp)))

(define (exp-checker type)
  (lambda (exp) (and (pair? exp) (eq? (car exp) type))))

(define quote-exp? (exp-checker 'quote))
(define if-exp? (exp-checker 'if))
(define lambda-exp? (exp-checker 'lambda))

(define (substitute exp params args bound)
  (cond ((constant? exp) exp)
    ((symbol? exp)
     (if (memq exp bound)
         exp
         (lookup exp params args)))
    ((quote-exp? exp) exp)
    ((lambda-exp? exp)
     (list 'lambda
           (cadr exp)
           (substitute (caddr exp) params args (append bound (cadr exp)))))
    (else (map (lambda (subexp) (substitute subexp params args bound))
           exp))))

(define (lookup name params args)
  (cond ((null? params) name)
    ((eq? name (car params)) (maybe-quote (car args)))
    (else (lookup name (cdr params) (cdr args)))))

(define (maybe-quote value)
  (cond ((lambda-exp? value) value)
    ((constant? value) value)
    ((procedure? value) value)  ; real Racket primitive procedure
    (else (list 'quote value))))
```