CISC280 Concluding Learning Experience, Spring 2006

NAME _____

General instructions:

Turn off all noise-making electronic devices, such as cell phones. Disturbance by such a device during the exam may result in a penalty not to exceed a full letter grade.

You may leave the classroom once the exam has begun, but you may not return.

There are 9 pages worth a total of 123 points. Read the problems very carefully. Identify what kind of answer the problem asks for. If writing a procedure, carefully look at requirements for input and output, and any restrictions on how it must be written or which other procedures may be used.

Be sure you understand what the question is before you answer it. Before you answer the question, be sure you understand what is being asked. Asked know, before answer give. Answer asked.

You may assume a list argument will be flat unless it is otherwise specified, and that the elements will not produce errors for the procedures described.

Do not do unnecessary testing. For example, testing for both list? and null? instead of using one test and then else would be considered unnecessary testing.

Do not make code unnecessarily inefficient. An extra procedure call here or there is ok, but do not make an $O(n)$ problem into $O(n^2)$.

Do problems you are confident about first. If you finish the problems you know, write what you do know about other problems to gain partial credit; but erroneous information may detract from that credit, so don't make stuff up.

You may use any of the Scheme primitives we use in class.

1. (5 pts) Write the procedure last-pair that returns a pointer to the last cons in a proper list. (It is an error to call last-pair on a null list.)

2. (5 pts) Show the box-and-pointer representation of memory that results from evaluating these three expressions.

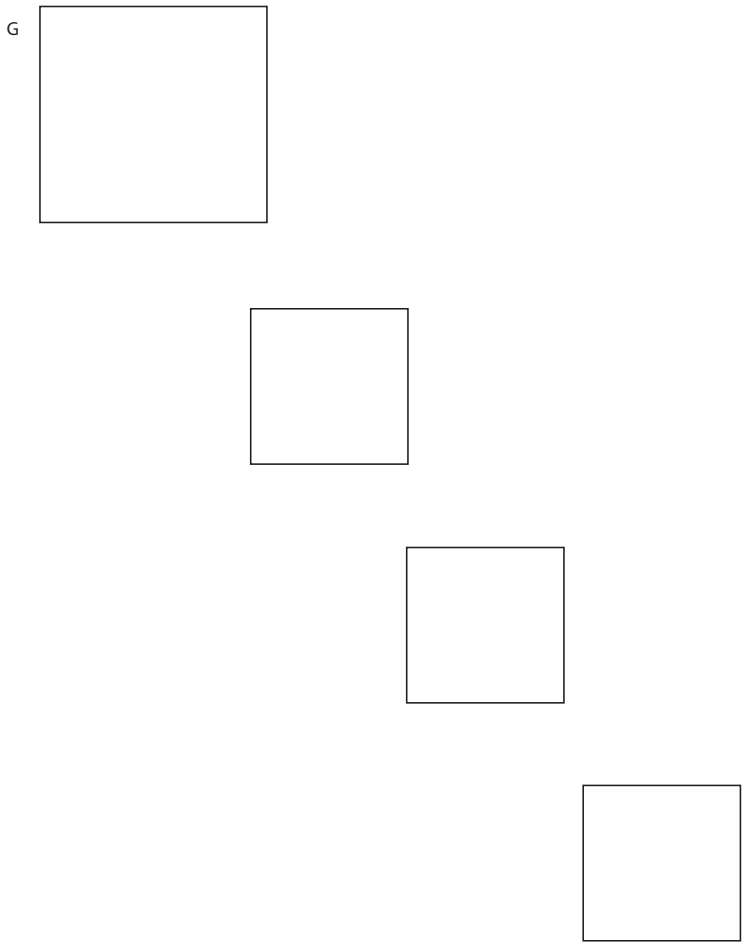
```
(define a (list (list 2 3) 4))
(define b (list 5 6))
(define c (append a b))
```

3. (5 pts) Show the box-and-pointer representation of memory that results from evaluating these three expressions.

```
(define a (list (list 2 3) 4))
(define b (list 5 6))
(append! a b)
```

4. (7 pts) Evaluate the three expressions shown and fill in the diagram. Only use frames provided (you may not need all of them).

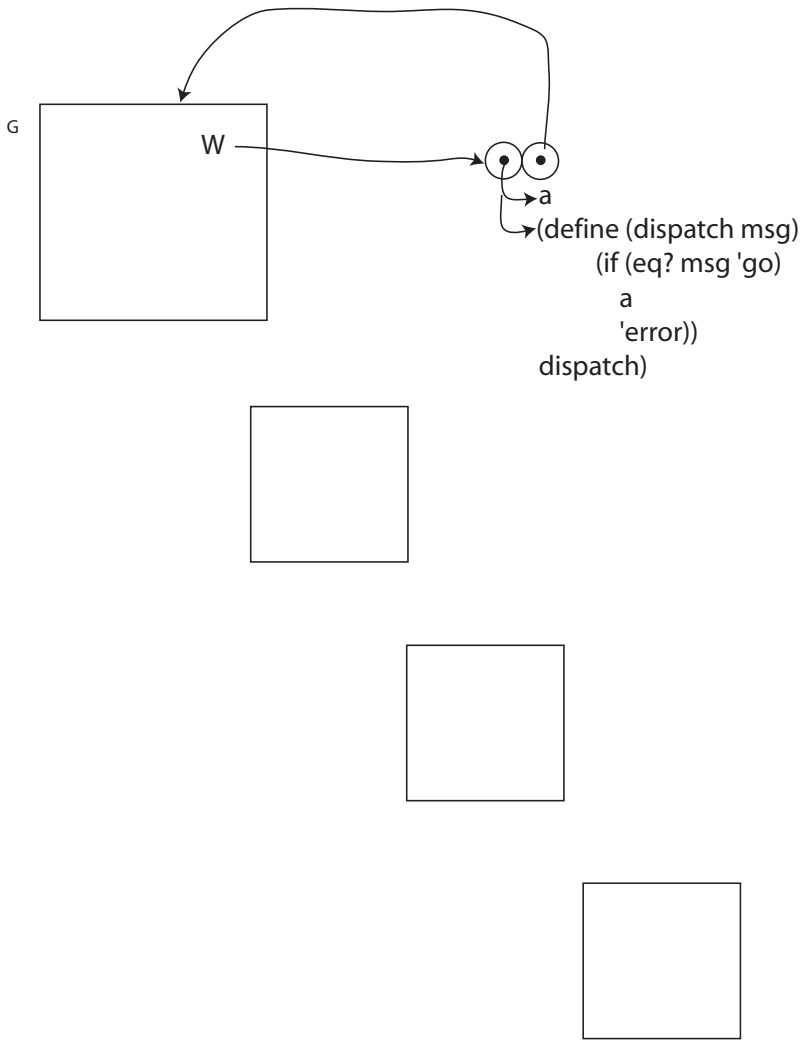
```
(define x 3)
(define (f x) (set! x (* x 2))
              (lambda (x) (+ x 2)))
(if (> x 2)
    (set! x 5)
    (set! x (f x)))
```



5. (14 pts) Evaluate

```
(define a 7)
(define t (w 10))
(set! a (t 'go))
```

to fill in the diagram. Only use frames provided (you may not need all of them).



6. (6 pts) Complete this function definition.

```
(define (reverse elts)
  (define iter (lambda (alist blist)
                 (cond ((null? alist) blist)
                       (else



  (iter elts ()))
```

7. (6 pts) Complete this function definition.

```
(define reverse (lambda (alist)
  (cond ((null? alist) ())
(else
```

8. (8 pts) Compare the previous two functions for O(), both time and space.

Don't write below this point.

Your task is to finish an implementation of the special form **and** in the mc-eval. The specification from the SICP text is:

> and: The expressions are evaluated from left to right. If any expression evaluates to false, false is returned; any remaining expressions are not evaluated. If all the expressions evaluate to true values, the value of the last expression is returned. If there are no expressions then true is returned.

You are given the following code:

```
(define (and? exp) (tagged-list? exp 'and))

(define (and-exps exp) (cdr exp))

(define (last-exp? seq) (null? (cdr seq)))
(define (no-exp? seq) (null? seq))
(define (first-exp seq) (car seq))
(define (rest-exps seq) (cdr seq))

(define (eval-and exps env)
  (cond ((last-exp? exps)
         (if (true? (eval (first-exp exps) env))
             (eval (first-exp exps) env)
             'false))
        ((true? (eval (first-exp exps) env))
         (eval-and (rest-exps exps) env))
        (else 'false)))
```

9. (5 pts) Show the new line of code that would need to be added to **eval** to implement this code. What line in eval should the new line follow?

10. (5 pts) The code does not meet spec. Show the case you need to add to meet spec, and say where it goes.

The new **and** always seems to be correct with regards to truth values, but you notice some strange behavior when one of the expressions includes assignment:

11. (5 pts) Show a specific **and** expression that will cause undesirable behavior:

12. (5 pts) Show the replacement code for one whole case in the **cond** to fix the undesirable behavior. Your fix should make the code simpler, not more complex.

The next two questions are about the fundamental difference between eval-sequence and list-of-values that we discussed in class.

13. (7 pts) What is the difference? Why do we need both?

14. (3 pts) Show a single example argument that can be passed to both procedures, but returns a different result from each one. Assume the environment does not change.

15. (2 pts) You are managing eleven hospital databases as part of a single system, and you need to choose a method (possibly not discussed in class) for implementing generic operations. Suppose that you think every hospital will need to perform radically different operations on records, and you expect new hospitals (with new record structures) to join your system. Which hospital food would you prefer: the chicken parmesan or the vegetable lasagna? Why? What about Jello?

16. (15 pts) What software do you modify or write to to make each of the following changes to a generic type system? Specify what "piece" you write or what you would modify and how. Your answers should be very brief.

    (a) Add a new type to a data-directed programming system

    (b) Add a new type to a dispatching-on-type system

    (c) Add a new type to a message-passing system

Consider the problem from lab 10:

```
(define (make-person name birthplace threshold)
  (let ((mobile-obj (make-mobile-object name birthplace))
          . . .
        )
    (lambda (message)
      (cond . . .

            ((eq? message 'install)
             (lambda (self)
               (add-to-clock-list self)
               ((get-method mobile-obj 'install) self) ))   ; **
       ))))

(define (make-mobile-object name place)
  (let ((named-obj (make-named-object name)))
    (lambda (message)
      (cond
          . . .
            ((eq? message 'install)
             (lambda (self)
               (ask place 'add-thing self)))

            ...))))
```

Louis Reasoner suggests that it would be simpler if we change the last line of the **make-person** version of the **install** method to read:

```
               (ask mobile-obj 'install) ))   ; **
```

Alyssa P. Hacker points out that this would be a bug.

You drew diagrams and wrote explanations to show this was so in your lab. Now, I want you to use the facilities provided in the adventure game (i.e. don't use procedures that users aren't supposed to use) to demonstrate the contradiction Louis' suggestion creates. Assume the places **Smith** and **Purnell** are already created.

Write the following instructions as you would enter them in the interpreter:

(a) (4 pts) Create a person X in Smith.

(b) (4 pts) Move that person to Purnell.

(c) (6 pts) When the person moves, there will be an error message generated. Write it here and say where it comes from.

(d) (6 pts)Ask the right two "questions" to prove that X is now apparently in two places at once.

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))

(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
           (procedure-body procedure)
           (extend-environment
             (procedure-parameters procedure)
             arguments
             (procedure-environment procedure))))
        (else
         (error
          "Unknown procedure type -- APPLY" procedure))))

(define (list-of-values exps env)
  (if (no-operands? exps)
      '()
      (cons (eval (first-operand exps) env)
            (list-of-values (rest-operands exps) env))))

(define (eval-if exp env)
  (if (true? (eval (if-predicate exp) env))
      (eval (if-consequent exp) env)
      (eval (if-alternative exp) env)))

(define (eval-sequence exps env)
  (cond ((last-exp? exps) (eval (first-exp exps) env))
        (else (eval (first-exp exps) env)
              (eval-sequence (rest-exps exps) env))))
```