CISC280 Final Exam, 2003-2007

NAME _____

General instructions:

There are 18 questions worth a total of 116 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.

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 simple procedures we use regularly in class, and the following procedures (unless you are asked to define them):
apply
append
reverse
map
accumulate
filter
enumerate
eq?
equal? (only if needed)
set!, set-car!, set-cdr! (only if needed)

1. (6 pts) What are the three methods shown in class to evaluate a sequence of expressions in Scheme?

2. (10 pts) How do objects, such as the message-passing bank accounts we created, differ from streams with respect to time?

The Meta-evaluator

See the attached meta-evaluator code as you answer the following questions.

3. (5 pts) How do you start the meta-evaluator?

4. (5 pts) Describe exactly what changes to the meta-evaluator you would make to add a new primitive operation, subtraction.

5. (5 pts) Write the function primitive-procedure-names that belongs in the meta-evaluator (see setup-environment).

6. (10 pts) Write the function list-of-values that belongs in the meta-evaluator (see eval).

Memory

7. (6 pts) Define a fcn last-pair that walks through a proper list and returns the terminating pair.

8. (10 pts) Use last-pair to write `append!`, which appends two lists.

9. (2 pts) What is the order (big O) of `append!` ? Be specific.

10. (2 pts) What is the order (big O) of simple recursive `append`? Be specific.

11. (5 pts) Define two lists and draw them using box and pointer notation. Then show the box and pointer list that would result from joining the lists with `append`.

12. (5 pts) Now show the box and pointer list that would result from joining the lists with `append!` and explain how it would be different.

13. (5 pts) Is there any advantage to using `append` vs. `append!` ? When would one be better than another?

Halt

For the halt proof, we assumed that we were given the working code for an imaginary function `(halt? <program> <input>)` that returns true if a program stops when given the input, and false otherwise. We then wrote

```
(define (try arg)
   (if (halt? arg arg) (spam)
      'cease))
```

14. (5 pts) Write a procedure `spam` that makes the proof work.

15. (5 pts) What argument would you call `try` on to derive a contradiction?

16. (5 pts)What is the significance of the halt proof to computer science?

Your answer may not go below this line.

Hex Game

Suppose a hex is a list `(<letter> <x> <y>)`. You choose a board representation where all hexes that have been played are stored in a single binary search tree. Compare hexes first by x-coord, then by y-coord, i.e. `(X 2 5)` is less than `(X 3 2)`, and `(X 2 5)` is less than `(X 2 6)`.

17. (15 pts) Write a procedure to find if a hex has been played by either player; it takes a hex and a tree as arguments and returns the hex (if found) or null. Write whatever supporting procedures you need. Use appropriate procedural abstraction to get full credit.

18. (10 pts)

    Discuss how the performance of this representation compares to storing the played hexes in a simple list. Be specific about the complexity and interaction between the game play and the board representation.

19. 10 pts. Let p1 and p2 be procedures that take single numeric arguments. Write a procedure `max-proc` that takes two such procedures and a single number, and returns the procedure that has the highest return value for that input.

20. 10 pts. Write a predicate (subset? alist blist) that returns true if and only if every member of alist is also a member of blist. Elements may be numbers or symbols. Write any supporting procedures you use.

21. 18 pts. Draw the environment model showing the results of evaluating this sequence. Use the enumerated frames, in order. Be sure to write notes, as I do in class, so that I can give partial credit. If a value is changed, simply cross out the previous value once.

```
(define x 7)
(define (f x y)
  (define (g x) (set! x 5))
  (+ x y))
(set! x (f 2 3))
```

22. 18 pts. Draw the environment model showing the results of evaluating this sequence. Use the enumerated frames, in order. Be sure to write notes, as I do in class, so that I can give partial credit. If a value is changed, simply cross out the previous value once.

```
;watch parens!
(define (f x y)
  (define (g) (set! x 5) x)
  (lambda (x) (+ x (g))))
(define x ((f 2 3) 7))
```

23. 25 pts. Breathe. Then write the procedure **eval** for the meta-circular evaluator. I do not expect your code to be perfect; you are trying to communicate your knowledge of the meta-circular evaluator. Your procedure names don't have to match those of the text, but they should clearly indicate that you understand what is happening in each case. Use comments to clarify if you aren't satisfied with your code. Hint: Do the other problems first, then go through the code you (and I) have written to be sure you have all (or most) of the cases you need.

a. Exactly how many times will ExpandPartialSolution be called?

b. 5 pts. What argument does eval need that apply does not? Why?

b. Refering to each square as a list of row and column, show the order of the squares chosen for expansion. Your answer should look like this: (7 6) (7 5) (4 5) ....

24. 20 pts. For the maze shown, and using the code given, answer the following questions:

25. a. 18 pts. Show what is returned by the interpreter for each of the following, then describe the result (or explain why it produces an error). If a message is returned (like #proc:f or #error), show the approximate message. If it is a procedure, describe what it does.

    Example:

    > (lambda (x) (+ x 1))
    result: #proc
    description: an unnamed procedure that returns argument + 1

    >(define ones (cons 1 (delay ones)))
    >ones
    result:
    description:

    >(cdr ones)
    result:
    description:

    >(define (ones) (cons 1 (delay (ones))))
    >ones
    result:
    description:

    >(force (cdr (ones)))
    result:
    description:

    >(define mystery (cons 1 (delay (+ 1 (car mystery)))))
    >mystery
    result:
    description:

    >(stream-cdr mystery)
    result:
    description:

b. 7 pts. Define a procedure that returns the infinite stream 0, 1, 2, ... You may not use any stream utilities.

c. 8 pts. Write the stream utility (stream-enumerate-interval $< low > < high >$).

26. a. 4 pts. Consider an efficient intersection procedure for ordered list representation of a set. What is the complexity, in big O notation?

b. 8 pts. How many times would it be called for the two sets $\{1, 2, 3\}$ $\{1, 3, 4\}$, assuming they were represented in the proper way? SHOW EACH of the calls to intersection, with arguments.

27. a. 4 pts. Draw the box and pointer notation for the structure (4 () 5 ((6) 7)).

b. 4 pts. Define (using Scheme code) a list of numbers with a cycle in it, so that when it is traversed, the numbers encountered are:
1, 2, 3, 2, 3, 2, 3, ...

c. 15 pts. Write a procedure **cycle?** that detects a cycle in a list without modifying the structure. The cycle may begin anywhere in the list.

d. 3 pts. What is the order of growth of space of the procedure in part c if the list is size n? Use big O

notation.

28. a. 10 pts. What are generic procedures? Why do we use them?

    b. 10 pts. What is message-passing?

29. Extra Credit:

    5 pts. Which of the following did Terry say (approximately) in the last class:

    "Recursion is always better than loop structures."

    "Sometimes loops are a better solution than recursion."

    "If Scheme had loops, it would no longer be semantically clean."

    "Most commercial Scheme compilers do have loop structures."

    and what was he referring to when he said it?

Scheme programming

30. (8 pts) Write the simple recursive procedure map to behave as follows (you may assume the first argument is a single procedure).

```
> (map (lambda (x) (+ 1 x)) '(1 2 3))
(2 3 4)
```

31. (4 pts) Write the recursive procedure “loop” that takes no arguments and never returns.

Complete the following procedure definitions.

32. (4 pts)

```
;Assume the lists are the same length
;> (interleave '(2 3 4) '(5 6 7)) -> (2 5 3 6 4 7)
(define interleave
  (lambda (alist blist)
    (cond ((null? alist) ())
          (else
```

33. (4 pts)

```
;Assume you have the primitives you need.
;Allows comparison of hexes for storage in a binary tree only.
(define (less-than-hex? h1 h2)
  (cond ((< (get-x-coord h1)(get-x-coord h2)) #t)
        ((> (get-x-coord h1)(get-x-coord h2)) #f)
        (else
```

34. (4 pts)

```
;(accumulate + 0 '(1 2 3)) -> 6
(define (accumulate op init alist)
  (cond ((null? alist) init)
        (else
```

35. (14 pts) Draw the environment diagram as shown in class. Use the frames in the order they are created, starting at upper left.

```
(define (f a b) (* a b x))
(define g (lambda (x)
            (set! x 7)
            (f 1 2)))
(define x 2)
(define z (g 6))
```

36. (14 pts) Assume *all* the following expressions are evaluated **in order**. Fill in the blanks with the value or message printed by the interpreter. If there is an error or other Scheme message, you do not have to be exact. If you get confused, try drawing a little environment diagram. :)

```
> (define (f x)
     (/ x 2)
     x)
> (define a 20)
> (f a)
```

a. _____

```
> (f a)
```

b. _____

```
> (define b 8)
> (define (g x)
     (set! x (* x 2))
     x)
> (g b)
```

c. _____

```
> (g b)
```

d. _____

```
> (define (h x)
     (define (k)
        (set! x (/ x 2))
        x)
   k)
> (define k (h 100))
> (k)
```

e. _____

```
> (h 100)

f. _____

> (k)

g. _____
```

Generic Operations

37. (4 pts) Write a **single call** to the apply-generic procedure below that will perform the `mult` operation on two numeric arguments, each tagged as type `blt`.

```
(define (apply-generic op . args)
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (if proc
          (apply proc (map contents args))
          (error
            "No method for these types -- APPLY-GENERIC"
            (list op type-tags))))))


>
```

38. (4 pts) Draw a very small, simple table that would enable that call to work. Show ONLY column and row headings that are NECESSARY, as well as contents (as I showed them in class).

39. (3 pts) What two structures did we discuss that can help with the combinatorial explosion of types?

40. (2 pts) What is it called when the interpreter makes use of those structures to help with the multitude of argument types by affecting the types of arguments?

SHORT Answers: For the following questions, focus on coding aspects that are characteristic of the approach mentioned. Read carefully!

41. (3 pts) Describe in general terms what you write/modify to add a new *operation* to a dispatching-on-type system.

42. (3 pts) Describe in general terms what you write/modify to add a new *operation* to a data-directed type system.

43. (3 pts) Describe in general terms the piece or pieces of code you write to add a new *type* to a message-passing type system.

44. (3 pts) In data-directed type systems, how did we avoid procedure name conflicts between procedures without changing the names?

Evaluation

45. (12 pts) In the meta-circular evaluator, when a cond expression is passed to eval it is transformed *before* it is recursively evaluated. Follow the code in the evaluator and show the transformed version of the following cond expression.

```
(cond ((> x 2) (f x))
      ((> x 3) (g x) x)
      (else 7))
```

46. Under one of the models we used to describe evaluation, this code is ambiguous. Which model?

```
(define test
  (lambda (x)
    (set! x (+ 1 x))
    x))
```

(3 pts)_____

47. What property of this model makes the code ambiguous?

(3 pts)_____

48. One procedure in the meta-circular evaluator is responsible for ensuring that this code segment from 46 is not ambiguous (assuming all necessary primitives are defined). What is the name of the procedure?

(4 pts)_____

Project

49. (5 pts) When bridges were added to the project, was it necessary (as discussed in class) to change how the AI rated moves based on clusters? Explain **briefly** why or why not.

Box and Pointer

50. (4 pts)

```
> (define a (list 1 2 3))
> (set-cdr! (last-pair a) a)
```

Draw the box and pointer notation for a.

51. (2 pts)

```
> (define b (list 1 2 3))
> (set-cdr! (last-pair b) (car b))
```

Draw the box and pointer notation for b.

52. (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.)

53. (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))
```

54. (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)
```

55. (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)))
```

56. (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).

57. (6 pts) Complete this function definition.

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




  (iter elts ())))
```

58. (6 pts) Complete this function definition.

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

59. (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)))
```

60. (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?

61. (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:

62. (5 pts) Show a specific **and** expression that will cause undesirable behavior:

63. (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.

64. (7 pts) What is the difference? Why do we need both?

65. (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.

66. (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?

67. (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

```
(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))))
```

68. (15 pts) Evaluate the expressions shown and fill in the diagram. Only use frames provided (you may not need all of them).

```
(define x 7)
(define y 5)
(define f
  (lambda (x) (* x y)))
(define z (f 4))
```

69. (15 pts) Evaluate the expressions shown and fill in the diagram. Only use frames provided (you may not need all of them). Hint: **if** is not a procedure application.

```
(define a 5)
(define (f x y)
  (if (= y 4)
      1
      (f (* x y) (- y 1))))
(define b (f a a))
```

70. (16 pts) Evaluate the expressions shown and fill in the diagram. Only use frames provided (you may not need all of them).

```
(define x 5)
(define (f x)
  (lambda (z)
    (set! x 3)
    (* z x)))
(define z ((f x) x))
```

71. (10 pts) When we studied assignment we examined bank account simulators that changed the value of a variable. Define a procedure (**make-count-down n**). It returns a procedure which decrements a variable (initial value is n). It returns `#f` if the variable hasn't reached zero, returns `#t` if it has. For example, after you have defined make-count-down, you could have the following:

```
> (define counter (make-count-down 3))
> (define (watcher) (if (counter) 'boom! 'not-yet))
> (watcher)
not-yet
> (watcher)
not-yet
> (watcher)
not-yet
> (watcher)
boom!
```

72. (8 pts) Draw the box-and-pointer diagram that results from evaluating the expressions:

```
(define a (list 1 (list 5) 2))
(define b (list 3 4))
(define c (append a b))
```

73. (5 pts) Complete this function definition.

```
(define (last-pair elts)
  (cond


        (else (last-pair (cdr elts)))))
```

74. (5 pts) Write the procedure **append!** which takes two lists as arguments.

75. (5 pts) Draw the box-and-pointer diagram that results from evaluating the expressions:

```
(define a (list 1 2))
(define b (list 3 4))
(define c (append! a b))
```

76. (5 pts) Complete this function definition.

```
;This function displays k spaces
(define (indent k)
  (cond ((eq? k 0) ())
```

77. (5 pts) Complete the function definition below. In this example, 5 is the root and 2 is 5's left child.

```
> (print-tree t1 3)
        8
    7
        6
 5
    2
        1
()

;This function uses indent to print a tree. Use the tree
;functions we wrote in class.
(define (print-tree tree k)
  (cond ((empty? tree) ())
        (else
```

78. (6 pts) Complete this function definition.

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

Consider the functions f1 and f2.

```
(define g
  (lambda (a b n)
    (cond ((= n 1) a)
          (else (g b (+ a b) (- n 1))))))

(define h
  (lambda (n)
```

```
(cond ((= n 1) 1)
      ((= n 2) 1)
      (else (+ (h (- n 1)) (h (- n 2))))))
```

79. For each function draw THREE levels of the call tree generated by the call shown:

    (a) (6 pts) `(g 1 1 20)`

    (b) (6 pts) `(h 20)`

    (c) (4 pts) What is big O (time) for g?

    (d) (4 pts) What is big O (time) for h?

80. (4 pts) Linear recursive functions can often be made tail-recursive by adding _____ _____.

81. (4 pts) Tail-recursive functions do not have _____ _____ that wait for recursive calls.

Meta-circular evaluator

82. (4 pts) Xenon has a large file of code that uses map. Suppose Xenon wants to define a new, improved definition of map, and Xenon wants to be able to compare his/her new map with the old map when

Xenon uses her/his file. What single expression should Xenon type in the interpreter to be certain s/he can use both?

83. (4 pts) This problem must be handled in the meta-circular evaluator too. For what procedure is it necessary? (Note: this problem is not about map; it is about being able to use both old and new versions of a procedure.)

84. (6 pts) Potstkr performs an analysis of some code and finds that procedures are called far more frequently than procedures are created with lambda. Potstkr decides to improve the efficiency of **eval** by swapping the two cases, so that calls are handled sooner, and thus fewer clauses are evaluated. Will this change affect the way the eval function handles a cond expression? Explain.

85. (6 pts) Assume the mc-eval has been loaded, but is not running yet. Show the expression returned by the interpreter when given the expression:

```
> (expand-clauses '(((> 3 4) 5) ((< 3 4) 6)))
```

86. (4 pts) Why does **expand-clauses** contain calls to **sequence→exp**? Your answer should be in terms of your knowledge of the Scheme language.

87. (6 pts) What changes would have to be made to the **eval** function to convert the syntax of the interpreted language to infix notation, e.g. (2 + 3)? Explain briefly.

88. (18 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 generic operation to a dispatching-on-type system

(b) Add a new generic operation to a data-directed programming system

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

89. Extra Credit: Which two functions that we've written/used in class did Joel Spolsky say were essential to the development of Google?

90. Extra Credit: Give one example of a viable commercial enterprise that Paul Graham says won in the marketplace because it used Lisp.