MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 1998, Final Exam

Your Name:

Open Book

Please write clear and concise answers to the questions in the spaces provided in this booklet. You may use scratch paper if you need, but the spaces we provide are the **only** places we will look at when grading.

Circle Your Recitation Section (2 points)					
Time	Instructor	T.A.	Time	Instructor	T.A.
9	Berthold Horn	Christiana Toutet	10	Berthold Horn	Vinay Pulim
10	Gerry Sussman	Derek Bruening	11	Hal Abelson	Ben Adida
10	Larry Rudolph	Jeremy Lin	11	Larry Rudolph	Tony Ezzat
10	Mike Leventon	George Dolina			
11	Paul Viola	Barbara Cutler	12	Paul Viola	Calista Tait
11	Madhu Sudan	Shantanu Sinha	12	Madhu Sudan	Pedro Zayas
12	Eric Grimson	Ilya Shlyakhter	1	Jim Miller	Chris Tserng
1	Duane Boning	Mike Allen	2	Duane Boning	Alex Lee

Please do not write below this line:

Problem	Value	Grade	Grader	Problem	Value	Grade	Grader
0	2			6	22		
1	12			7	12		
2	20			8	10		
3	21			9	23		
4	14			–	–	–	–
5	14			Total	150		

Question 1 (12 points):

Each of the parts below should be treated as independent. For each part, a sequence of expressions is given, which you may assume is typed to a Scheme interpreter and evaluated in the order shown. Write the value that will be printed in response to the **last expression** in each sequence.

Part a:

```
(define square (lambda (x) (* x x)))

(define f
   (lambda (g)
      (lambda (f)
        (f g))))

((f 5) square)
```

Part b:

```
(define stream
  (cons-stream 1
     (stream-map (lambda (x) (expt 2 x))
                 stream)))

(stream-car
   (stream-cdr
      (stream-cdr
         (add-streams stream (stream-cdr stream)))))
```

Part c:

```
(define x (list 'x 'y))

(define y (list x 'x 'z))

(set-cdr! (cdr x) (cdr y))

y
```

Question 2 (20 points):

For each of the following statements, indicate whether the statement is true or false by circling T or F:

T F Independent creation is a valid legal defense against claims of copyright infringement.

T F The main reason Scheme requires garbage collection is that procedures can be recursive.

T F The time required to perform a stop-and-copy garbage collection grows as the amount of memory in actual use, not the total amount of memory available to Scheme.

T F It is possible to write a Scheme procedure, `safe?`, with the behavior that, for any procedure `proc` and list of arguments `args`, `(safe? proc args)` will return `true` if `(apply proc args)` returns a value, and will return `false` if `(apply proc args)` either signals an error or enters an infinite loop.

T F Using the AMB evaluator, the possible values of `(+ (amb 0 1) (amb 0 1))` are the same as the possible values of `(* 2 (amb 0 1))`.

T F PICS is a system developed by the World Wide Web Consortium for attaching content labels to Web pages.

T F If A and B are two events in a concurrent system, it is always possible to deduce from the program that either A happens before B, A and B happen simultaneously, or A happens after B.

T F Quantum computing can be faster than classical computing because qubits can have the values 0, 1/2, or 1 while classical bits can only have the values 0 or 1.

T F Suppose Alice and Bob have never communicated before today. Suppose they now start communicating, and that Eve can hear everything that Alice and Bob say to each other. Then any information that will be shared by Alice and Bob will be known to Eve.

T F Image-guided surgery is a promising new technique being developed at the MIT Artificial Intelligence Lab, which will start being tried with real patients next yer.

Question 3 (21 points):

In parts a, b, and c of this problem, (on the next few pages), you will be adding a special form to the Metacircular Evaluator. This special form is an `until` expression, for example:

```
(until (>= i limit)
   (process i)
   (set! i (+ i 1))
   i)
```

An `until` expression consists of an end test (in the example, the expression `(>= i limit)`) and a body, which is a sequence of expressions (in the example,r the sequence `(process i) (set! i (+ i 1)) i`). The evaluation of an `until` form should occur in the following manner:

- First the body is evaluated. The body's value is the value of the last expression in the body.
- Next the end test is evaluated.
- If the value of the end test is true, then the value returned for the `until` expression is the value computed when evaluating the body in the previous step.
- If the end test is false, then the body is evaluated again, and this continues until the end test evaluates to true.

In the above example, the expression would ultimately return the value of `limit`, assuming that `i` had a value no larger than `limit` before the `until` expression was evaluated.

Part a: We need a data abstraction for dealing with `until` expressions.

Define the predicate `until?`.

Define the selectors `until-body` and `until-test` which should respectively return the body and end test of a complete `until` expression.

Part b: Assume that the procedure `eval-until` (which you will write shortly) handles the actual evaluation of an `until` expression. What changes should be made to the `eval` procedure to allow it to recognize `until` special forms? Any such change should be consistent with other parts of `eval`.

Part c: Write the procedure `eval-until`.

Question 4 (14 points):

You are to add the `until` special form of Question 3 to the Explicit-Control Evaluator. To the `eval-dispatch` part of the evaluator, we add

```
(test (op until?) (reg exp))
(branch (label ev-until))
```

Complete the code at the label `ev-until` by filling in the blanks below with the appropriate instructions.

```
ev-until
  (save continue)

  (save exp)                          ; save the entire expression

  (assign unev (op until-test) (reg exp))    ; get the end test

  (save unev)                         ; save it for later


  (assign ________________ (op until-body) (reg exp))   ; get the body

  (assign continue (label ev-after-until-body)) ; where to go when done

  (save env)                          ; save the environment

  (save continue)

  (goto ______________________)       ; evaluate the body


ev-after-until-body   ; now need to evaluate the end test

  (restore env)


  ______________________________


  ______________________________


  ______________________________
```

```
  (save env)

  (assign continue (label ev-after-until-test))


  (goto ________________________________)  ; evaluate the end test

ev-after-until-test  ; check to see if end test is true

  (restore env)

  (test (op true?) ____________________________)

  (branch (label done-until))   ; branch if end test true


  _______________________________________


  _______________________________________


  _______________________________________

  (goto (label ev-until))

done-until  ; clean up when done


  _______________________________________


  _______________________________________


  _______________________________________


  (goto ________________________)
```

Question 5 (14 points):

The following controller code was produced by the compiler of Problem Set 12. We have added the line numbers for referencing purposes only.

```
1   ((assign val (op make-compiled-procedure) (label entry10) (reg env))
2    (goto (label after-lambda9))
3   entry10
4    (assign env (op compiled-procedure-env) (reg proc))
5    (assign env (op extend-environment) (const (x y f g)) (reg argl) (reg env))
6    (assign proc (op lookup-variable-value) (const f) (reg env))
7    (save continue)
8    (save proc)
9    (assign val (const 3))
10   (assign argl (op list) (reg val))
11   (save env)
12   (save argl)
13   (assign proc (op lookup-variable-value) (const g) (reg env))
14   (assign val (op lookup-variable-value) (const y) (reg env))
15   (assign argl (op list) (reg val))
16   (test (op primitive-procedure?) (reg proc))
17   (branch (label primitive-branch13))
18  compiled-branch12
19   (assign continue (label after-call11))
20   (assign val (op compiled-procedure-entry) (reg proc))
21   (goto (reg val))
22  primitive-branch13
23   (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
24  after-call11
25   (restore argl)
26   (assign argl (op cons) (reg val) (reg argl))
27   (restore env)
28   (assign val (op lookup-variable-value) (const x) (reg env))
29   (assign argl (op cons) (reg val) (reg argl))
30   (restore proc)
31   (restore continue)
32   (test (op primitive-procedure?) (reg proc))
33   (branch (label primitive-branch16))
34  compiled-branch15
35   (assign val (op compiled-procedure-entry) (reg proc))
36   (goto (reg val))
37  primitive-branch16
38   (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
39   (goto (reg continue))
40  after-call14
41  after-lambda9
42   (perform (op define-variable!) (const doit) (reg val) (reg env))
43   (assign val (const ok))
44   (goto (reg continue))
```

Part a: Consider lines 9 and 10. Write a Scheme expression whose value is the same as the value in `argl` after these lines are executed.

Part b: Consider lines 13 to 24. Write a Scheme expression whose compilation would produce these lines.

Part c: Consider lines 25 to 27, plus lines 9 to 12. Assume that `g` is bound to `(lambda (x) (* x x))` and that `y` is bound to `3`. Write a Scheme expression whose value is the same as the value in `argl` after these lines are executed.

Part d: Consider lines 25 to 29, plus lines 9 to 12. Assume that `x` is bound to `27`. Write a Scheme expression whose value is the same as the value in `argl` after these lines are executed.

Part e: Consider lines 6 to 39. Write a Scheme expression whose compilation would produce these lines.

Part f: Write a Scheme expression whose compilation would produce the entire code.

Question 6 (22 points):

Consider the following expression:

```
(define (make-thing init)
  (let ((value init)
        (previous '())
        (next '()))
    (lambda (m)
      (cond ((eq? m 'value) value)
            ((eq? m 'init) init)
            ((eq? m 'increment)
             (lambda (change)
                (set! value (+ value change))))
            ((eq? m 'reset)
             (set! value init))
            ((eq? m 'previous)
             previous)
            ((eq? m 'set-previous)
             (lambda (th)
               (set! previous th)))
            ((eq? m 'next)
             next)
            ((eq? m 'set-next)
             (lambda (th)
               (set! next th)))))))

(define trial (make-thing 10))
```

Part a: Assume that the above two expressions are evaluated in the global environment. Draw an environment diagram that represents the environment created during the evaluation of these expressions. You need not write out the entire body of any procedure objects in your diagram.

Part b: Now suppose that we have a large collection of "things" constructed by make-thing. For simplicity we will assume that each has a unique value. We want to assemble this collection into a linear list, ordered by value. The idea is that if a thing has a `next` neighbor, then the thing's value is less than `next`'s. Similarly, if a thing has a `previous` neighbor, then the thing's value is greater than `previous`'s.

We will connect all the things together by using the following (partially incomplete) procedure.

```
(define (ripple new current)
  ;; idea is to insert in right place by moving left or right
  (cond ((> (new 'value) (current 'value))
         <exp1>)
        ((null? (current 'previous))
         <exp2>)
        ((> (new 'value) ((current 'previous) 'value))
         <exp3>)
        (else <exp4>)))
```

The idea is that we can take an unconnected thing (`new`) and a set of connected things, and insert the unconnected thing by "rippling" it with any member of the connected set (by picking any one of them, `current`). We can do this for each unconnected thing until we have connected all of them. When we are done, each thing should be bidirectionally linked, that is it should point to both its previous and next elements, if any.

The first thing we need to do is create a procedure to establish a bidirectional connection between two things. `(Connect from to)` should mutute `from` so that its `next` variable points to `to`, and should mutate `to` so that its `previous` variable points to `from` – i.e. make a connection from `from` to `to`.

Complete the definition below:

```
(define (connect from to)
```

Now using `connect`, complete the definition for `ripple`. For each piece below, you may need to write more than one expression. Don't worry about what value is returned by `ripple`.

```
(define (ripple new current)
  (cond ((> (new 'value) (current 'value))
         <exp1>)
        ((null? (current 'previous))
         <exp2>)
        ((> (new 'value) ((current 'previous) 'value))
         <exp3>)
        (else <exp4>)))
```

What code is needed in place of `<exp1>`?

What code is needed in place of `<exp2>`?

What code is needed in place of `<exp3>`?

What code is needed in place of `<exp4>`?

Question 7 (12 points): Many useful mathematical functions can be expanded into what is known as a power series, a sum of terms of increasingly higher-order powers of the function's argument. For example, we have

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots$$

We are going to approximate exponentiation by adding up terms in this series. Thus our first approximation to e^x would be

$$0$$

and our subsequent approximations would be

$$1$$

$$1 + x$$

$$1 + x + \frac{x^2}{2!}$$

$$1 + x + \frac{x^2}{2!} + \frac{x^3}{3!}$$

$$1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!}$$

and so on.

We are going to do this using streams.

Assume that the functions `add-streams`, `div-streams`, `mul-streams` and `scale-stream` are provided.

```
(define (add-streams s1 s2)
  (stream-map + s1 s2))

(define (div-streams s1 s2)
  (stream-map / s1 s2))

(define (mul-streams s1 s2)
  (stream-map * s1 s2))

(define (scale-stream stream factor)
  (stream-map (lambda (x) (* x factor)) stream))
```

The following definition will be of use to you.

```
(define ones (cons-stream 1 ones))

(define integers
  (cons-stream 1
             (add-streams ones integers)))
```

Part a:

Define `factorials` to be an infinite stream of factorials. Assume that the first few elements of this stream should be $1, 1, 2, 6, 24, \ldots$.

Part b: We need to generate a stream of powers of x, for any x. To do this, we will create a procedure `powers` which should behave as follows:

```
(display-stream (powers 2))
1
2
4
8
16
32
...
```

Here is a partial definition of `powers`:

```
(define (powers x)
  (define pwrs
    (cons-stream 1
          (scale-stream <exp1> <exp2>)))
  pwrs)
```

Provide the expression needed in place of `<exp1>` to complete this definition

Provide the expression needed in place of `<exp2>` to complete this definition

Part c: We can use the pieces defined above to create a stream of terms for a power series. The procedure `(exp-terms 2)`, for example, should return the sequence of values

$$\frac{1}{1}$$

$$\frac{2}{1!}$$

$$\frac{2^2}{2!}$$

$$\frac{2^3}{3!}$$

$$\frac{2^4}{4!}$$

and so on.

```
(define (exp-terms x)  <exp>)
```

Provide the expression needed in place of `<exp>` to complete this definition.

Part d: Using this, complete the definition below so that `exp-approx` will return a stream of successively better approximations to e^x, as listed at the beginning of this problem.

```
(define (exp-approx x)
  (define approx
     (cons-stream 0 <exp1>))
  approx)
```

Provide the expression needed in place of `<exp1>` to complete this definition:

Question 8 (10 points): The following is intended to count the number of pairs in a list structure.

```
(define (count-pairs x)
  (if (not (pair? x))
      0
      (+ (count-pairs (car x))
         (count-pairs (cdr x))
         1)))
```

This procedure will **NOT** correctly count the number of pairs. Draw box-and-pointer diagrams of list structures with exactly three pairs, for which this procedure would behave as follows.

Part a: Procedure would return 3.

Part b: Procedure would return 4.

Part c: Procedure would return 7.

Part d: Procedure would never return.

Question 9 (23 points):

Consider the following procedure for dealing with trees:

```
(define (tree-manip tree init leaf first rest accum)
  (cond ((null? tree) init)
        ((not (pair? tree)) (leaf tree))
        (else (accum
                (tree-manip (first tree) init leaf first rest accum)
                (tree-manip (rest tree) init leaf first rest accum)))))
```

Suppose that we provide a test tree:

```
(define test-tree '(1 (2 (3 (4) 5) 6) 7))
```

For each of the following parts, write the additional expressions needed so that evaluating `tree-manip` with the argument `test-tree` will accomplish each of the following:

Part a: Take the product of the even-valued leaves of the tree, which for `test-tree` should result in:

```
;Value: 48
```

```
(tree-manip test-tree
```

Part b: Flatten a tree, which for `test-tree` should result in:

```
;Value: (1 2 3 4 5 6 7)
```

```
(tree-manip test-tree
```

Part c: Deep-reverse a tree, which for `test-tree` should result in:

```
;Value: (7 (6 (5 (4) 3) 2) 1)
```

```
(tree-manip test-tree
```

Part d: Create a new tree, which keeps the odd-valued leaves of the original tree within the same tree structure, but completely removes even-valued leaves. For `test-tree` this should result in:

```
;Value: (1 ((3 5)) 7)
```

```
(tree-manip test-tree
```