

MASSACHUSETTS INSTITUTE OF TECHNOLOGY  
 Department of Electrical Engineering and Computer Science  
 6.001—Structure and Interpretation of Computer Programs  
 Spring Semester, 2002

**Final Exam**

**Closed Book – three sheets of notes**

Separately, we have distributed an answer sheet. You may use the space on the exam booklet for whatever temporary work you find useful, but you **MUST** enter your answers into the answer sheet. **Only this will be graded.** Each problem that requires an answer has been numbered. Place your answer at the corresponding number in the answer sheet.

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.

The answer sheet asks for your section number and tutor. For your reference, here is the table of sections.

Section	Time	Location	Rec. Instructor	Tutors
1	9:00	26-314	Konrad Tollmar	Pascal Rettig
2	10:00	34-301	Joel Moses	Todd Atkins
3	10:00	26-314	Konrad Tollmar	Pascal Rettig/Selim Temizer
4	10:00	34-304	Bruce Tidor	Ben Vandiver
5	11:00	34-301	Joel Moses	Todd Atkins/Selim Temizer
6	11:00	34-304	Bruce Tidor	Selim Temizer
7	12:00	34-301	Tony Eng	Chris Cheng
8	1:00	34-303	Jovan Popovic	Nicole Love
9	1:00	34-301	Tony Eng	Alex Andersson/Ben Vandiver
10	2:00	34-303	Jovan Popovic	Alex Andersson
11	11:00	26-314	Trevor Darrell	Leonid Taycher/Nicole Love
12	12:00	26-314	Trevor Darrell	Leonid Taycher

**IMPORTANT: Write your name on EVERY page of the answer sheet!**

Each part is independent. Please skim the exam before starting, in order to plan your time.

No grades for 6.001 will be available until after 2 PM on Friday, May 24. After that time, you may (1) wait until grades are mailed by the Registrar; (2) access your grades via WEBSIS (<http://student.mit.edu/>), or (3) contact the course secretary, Jill Fekete, in NE43-775. No grades will be given out by phone; you must appear in person with ID if selecting the third option.

Completed final exams will not be handed back to students, but will be available starting on May 24 for students to look at (but not remove from) Jill Fekete's office, Room NE43-775.

Please note: the Course VI undergraduate office DOES NOT HAVE 6.001 exam or course grades.

Good luck!

**Part 1 (20 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 into 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. If the sequence results in a error, write **error** and indicate the kind of error. If the final expression returns a procedure, write **procedure** and indicate the type of the procedure, e.g. `Integer`  $\mapsto$  `Integer`.

1.

```
(1 - 2)
```

2.

```
((lambda (a - b) (b a -)) 2 3 *)
```

3.

```
(define one 1)
(define two 2)
(define three 3)
(define list1 (list one 'two three))
(define list2 (list two list1 'three))
(set-cdr! (cadr list2) (caddr list1))
list2
```

4.

```
(define (stream-map proc str)
  (cons-stream (proc (stream-car str))
               (stream-map proc (stream-cdr str))))

(define (add-streams str1 str2)
  (cons-stream (+ (stream-car str1) (stream-car str2))
               (add-streams (stream-cdr str1) (stream-cdr str2))))

(define st
  (cons-stream 2
               (stream-map (lambda (x) (* x 2)) st)))

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

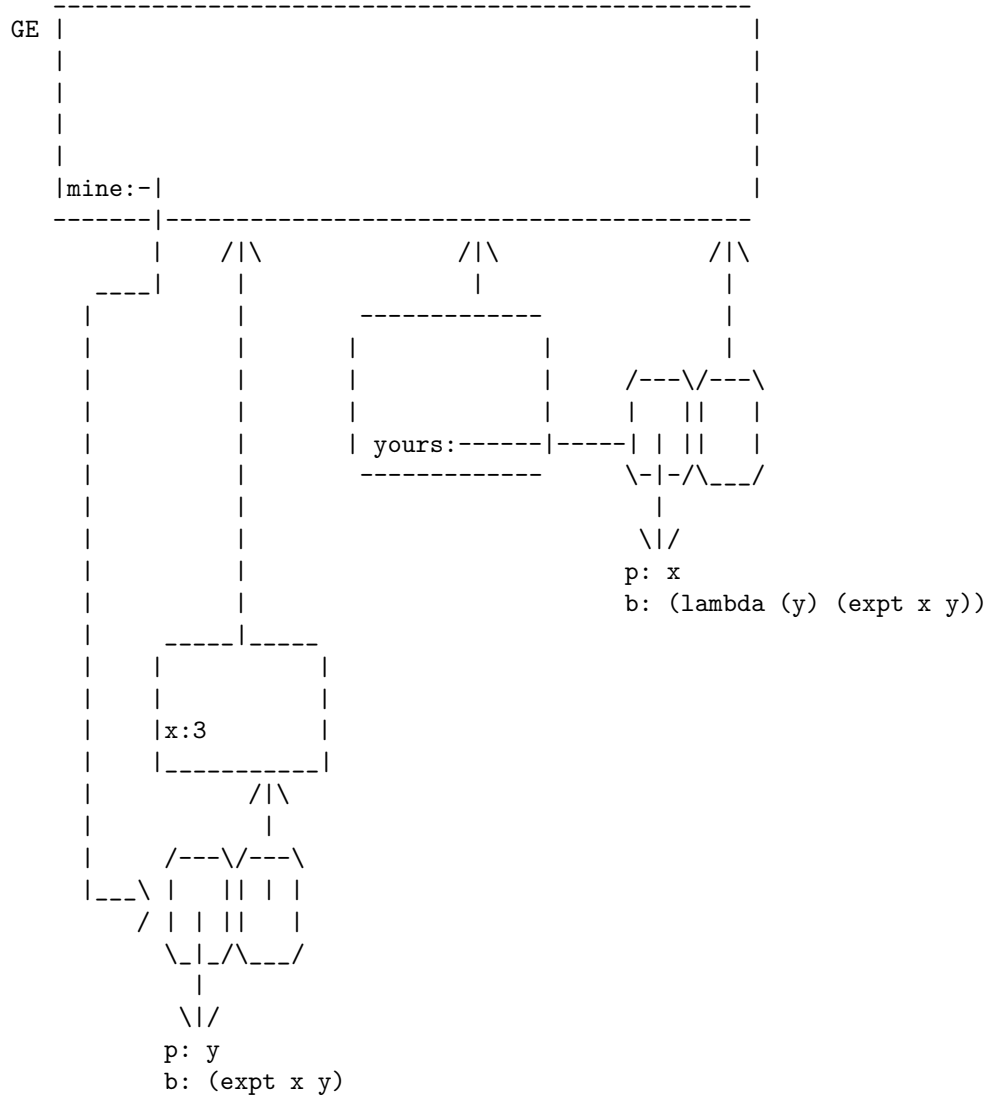
5.

```
(define p
  (lambda (p)
    (lambda (n)
      (n p))))

((p (lambda () (lambda (x) (* x y)))) (lambda (arg) arg))
```

**Part 2: (15 points)**

Consider the following environment diagram.



**6:** Write a single expression (without mutation or internal defines) whose evaluation would result in this environment diagram. If you wish to use a let expression, do not worry about whether the object associated with the sugared lambda is shown in the above diagram.

**Part 3: (20 points)**

In this question, you will be adding a special form to the Metacircular evaluator. For your convenience, a copy of the meta-circular evaluator code is included at the end of the exam, although you should be able to answer this question without needing to refer to it.

This special form is an `rbegin` expression. This expression acts much like a regular `begin` expression, the difference is that it evaluates its subexpressions in random order.

To implement this, we will use the following procedure:

```
(define (randomize exps)
  (define (aux old new n)
    (if (= n 0)
        (cons (car new) (append old (cdr new)))
        (aux (cons (car new) old)
              (cdr new)
              (- n 1))))
  (aux '() exps (random (length exps))))
```

We need a data abstraction for dealing with `rbegin` expressions.

**7: Define the predicate `rbegin?`, using the same pattern as the meta-circular evaluator.**

**8: Define the selector `rbegin-clauses` which should return the set of expressions within an `rbegin` expression.**

**9: Assuming one has extracted the clauses of an `rbegin` expression, define the selectors `rbegin-first` and `rbegin-rest` which should return the first and all the other clauses.**

Assume that the procedure `eval-rbegin` (which you will shortly complete) handles the actual evaluation of a `while` expression. Then the following clause can be added to `m-eval`:

```
((rbegin? exp)
 (eval-rbegin exp env))
```

Here is a partial implementation of `eval-while`.

```
(define (eval-begin exp env)
  (define (rbegin-loop clauses env)
    COMPLETE
  )
  (rbegin-loop (rbegin-clauses exp) env))
```

**10: Complete the procedure `rbegin-loop` by providing the missing code for `COMPLETE`. You may assume that the procedure `rbegin-no-clauses?` returns `true` if its argument has no clauses, and you may want to use `randomize`.**

**Part 4. (12 points)**

We want to add another new special form to our evaluator. This special form is called `let*` and has a different behavior from a normal `let`. For example, in

```
(define x 15)

(let ((x 25)
      (y x))
  (+ x y))
;Value: 40
```

the `let` would simultaneously bind `x` and `y` to values of their associated expressions, so in this case, `x` would be bound to 25 and `y` would be bound to the previously defined value of `x`, namely 15.

In `let*`, we evaluate each of the clauses in order, first binding `x` to 25, then binding `y` to the value of its expression relative to any previous bindings. Thus in this case `y` would be bound to 25, yielding a final value of 50, not 40.

One way to accomplish this is to create a sequence of environments in which each clause of the `let*` is handled in turn by binding the variable of that clause to the value of the associated expression. Thus we would bind the first variable to its value in an environment that extends the original environment, then we would bind the second variable to its value in an environment that extends the first, and so on.

To implement the evaluation of `let*` expressions, assume the following syntactic abstractions are available:

- `let*-clauses` takes a `let*` expression and returns the list of clauses that will bind local variables to values (in the example above, `((x 25) (y x))` is the list of clauses);
- `first-clause` takes a sequence of clauses, and returns the first one (in the example above this would be `(x 25)`);
- `rest-clauses` takes a sequence of clauses, and returns all but first one (in the example above this would be `((y x))`);
- `variable` takes a clause, and returns the variable part of the clause; and
- `value-exp` takes a clause, and returns the expression whose value is to be bound to the variable of the clause.
- `let*-body` takes a `let*` expression and returns the body of the `let*`, which is a sequence of expressions (in the example above this would be `((+ x y))`)
- `make-let` takes a set of clauses, and a body, and constructs a `let` expression, e.g.

```
(make-let '((x 25)) '((set! x (+ x 2)) (* x x)))
;Value: (let ((x 25)) (set! x (+ x 2)) (* x x))
```

We are going to add `let*` to our language by syntactically converting such an expression into an equivalent nested sequence of `let`'s. For the example above, we would convert the `let*` expression into

```
(let ((x 25))
  (let ((y x))
    (+ x y)))
```

then evaluate this expression with respect to the current environment.

Thus, to `meval` we add a clause

```
((let*? exp)
 (m-eval (convert-let* exp) env))
```

Here is a template for such a syntactic conversion:

```
(define (convert-let* exp)
  (define (doit clauses body)
    (if (null? clauses)
        <EXP1>
        (make-let <EXP2>
                  <EXP3>)))
  (doit <EXP4> <EXP5>))
```

11. Write a single expression for `<EXP1>`.
12. Write a single expression for `<EXP2>`.
13. Write a single expression for `<EXP3>`.
14. Write a single expression for `<EXP4>`.
15. Write a single expression for `<EXP5>`.

**Part 5: (16 points)**

In this part we are going to add a new form to the Explicit Control Evaluator, in this case the special form `or`. An `or` expression evaluates each of its arguments in order, from left to right. If one of the arguments evaluates to `true` then the process stops and the entire expression returns a `true` value. If all of the expressions are evaluated without this occurring, then a `false` value is returned.

To `eval-dispatch`, we add:

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

Below is a template of the code at the label `ev-or`, which you will be completing below. (The lines are numbered for convenience, but are not part of the register machine code.) You may assume the following data abstractions:

- `clauses` applied to an `or` expression returns all of the subexpressions of the `or`;
- `no-clauses?` returns `true` if there are not further clauses;
- `first-clause` returns the first subexpression of the expression;
- `rest-clauses` returns all of the subexpressions except the first one;

```
ev-or
1  (assign unev (op clauses) (reg exp))
2  (save continue)
3  (assign continue (label after-or-clause))
  or-loop
4  (test (op no-clauses?) (reg unev))
5  (branch (label end-clauses))
6  INSERT1
7  INSERT2
8  (save unev)
9  (save env)
10 (goto INSERT3)
  after-or-clause
11 INSERT4
12 INSERT5
13 (test (op false?) INSERT6)
14 (branch INSERT7)
15 (restore continue)
16 INSERT8
  end-clauses
17 (assign val (const false))
18 INSERT9
19 INSERT10
```

- 16: What single register machine instruction should be used for INSERT1
- 17: What single register machine instruction should be used for INSERT2
- 18: What should be used to complete the single register machine instruction at INSERT3
- 19: What single register machine instruction should be used for INSERT4
- 20: What single register machine instruction should be used for INSERT5
- 21: What should be used to complete the single register machine instruction at INSERT6
- 22: What should be used to complete the single register machine instruction at INSERT7
- 23: What single register machine instruction should be used for INSERT8
- 24: What single register machine instruction should be used for INSERT9
- 25: What single register machine instruction should be used for INSERT10

Suppose we decide to implement `and` in the Explicit Control evaluator. `And` has the opposite behavior: as soon as one subexpression is `false` it returns `false`, otherwise it returns `true`. Aside from changing the labels in the code you have just completed, you can implement `and` by simply changing parts of two lines.

**26: Indicate which lines should be changed, and what the change should be. You may assume that the predicates `true?` and `false?` can be used as primitive operations, and that the primitive values `true` and `false` exist.**

**Part 6: (12 points)**

Louis Reasoner wants to define a stream whose elements consist of consecutive numbers. He first attempts to define such a stream of numbers as follows:

```
(define incr
  (let ((val 0))
    (lambda ()
      (set! val (+ val 1))
      val)))
```

VERSION 1:

```
(define (make-stream)
  (cons-stream (incr) (make-stream)))

(define st (make-stream))
```

After he completes this version, he isn't certain that it is correct, so he also tries the following:

VERSION 2:

```
(define (make-stream)
  (cons-stream (incr) st))

(define st (make-stream))
```

He shows his work to Alyssa P. Hacker who suggests that he use `PRINT-STREAM-N` to examine the first few elements of each stream. Furthermore she suggests that he run his code on two different Scheme interpreters, one that implements lazy pairs using memoization, and one that does not.

Louis takes her advice, and, just to be sure, he prints his different versions of his streams twice. Shown below are pairs of printouts, of the sort that either version of the stream might have produced.

Possible outcomes:

```
(print-stream-n st 5)          ;;; OUTCOME A
1 1 1 1 1
```

```
(print-stream-n st 5)
1 1 1 1 1
```

```
(print-stream-n st 5)          ;;; OUTCOME B
1 2 3 4 5
```

```
(print-stream-n st 5)
1 2 3 4 5
```

```
(print-stream-n st 5)          ;;; OUTCOME C
1 2 3 4 5
```

```
(print-stream-n st 5)
6 7 8 9 10
```

```
(print-stream-n st 5)          ;;; OUTCOME D
1 2 3 4 5
```

```
(print-stream-n st 5)
1 6 7 8 9
```

```
(print-stream-n st 5)          ;;; OUTCOME E
1 1 1 1 1
```

```
(print-stream-n st 5)
2 2 2 2 2
```

```
ERROR                          ;;; OUTCOME F
```

List the outcome (chosen from A, B, C, D, E or F) that would be produced in each of the following cases, or indicate **none** if none of these outcomes is possible.

**27: Version 1; no memoization**

**28: Version 1; with memoization**

**29: Version 2; no memoization**

**30: Version 2; with memoization**

**Part 7: (18 points)**

For each of the following, select all of the statements that are true:

**31: A universal machine has the property that:**

- A: any process that can be computed by that universal machine can be computed by any other universal machine;**
- B: it can be used to implement the Scheme meta-circular evaluator;**
- C: it will always produce a legal output if given a syntactically valid input.**

**32: The Halting Theorem:**

- A: states that it is not possible to create a procedure that will always be able to determine if any procedure of no arguments is guaranteed to terminate its processing with an answer;**
- B: was first proved by Alan Turing;**
- C: only applies to interpreted languages like Scheme, but not compiled languages like C.**

**33: The property of “closure” as applied to an abstract data type implies:**

- A: that the result obtained by creating a compound data structure can itself be treated as a primitive object and thus be input to the creation of another compound object;**
- B: only applies to linear structures like lists, but not to complex structures like trees;**
- C: can be inherited by creating an abstraction that is based on a more primitive abstraction that has the closure property.**

**Part 8: (15 points)**

For each of the following questions, answer true or false. Grading on this part will be 3 points for each correct answer, -3 points for each incorrect answer, and 0 points for no answer.

**34: Executing compiled Scheme code will always involve at most as many stack operations as evaluating the same code in interpreted Scheme on the same machine.**

**35: All Abstract Data Types must have a constructor, selector(s) and mutator(s).**

**36: Both the Stop-and-Copy garbage collection algorithm and the Mark-Sweep garbage will fail to reclaim memory cells used by circular data structures (i.e., lists that contain pointers back to prior cells in the list) even if these structures are not reachable from any “root” of good memory.**

**37: A “higher order procedure” is a procedure that either takes a procedure as argument, or returns a procedure as value, or both.**

**38: The Explicit Control Evaluator requires at least seven registers (exp, env, proc, argl, unev, continue, val) in order to work properly.**

**Part 9 (16 points):**

The following code fragment was generated using the Scheme compiler.

39. What Scheme expression was compiled to generate line 11?
40. What Scheme expression was compiled to generate lines 8–21?
41. What Scheme expression was compiled to generate lines 36–47?
42. What Scheme expression was compiled to generate lines 30–60?
43. What Scheme expression was compiled to generate lines 8–61?
44. What Scheme expression was compiled to generate lines 1–62?
45. What Scheme expression was compiled to generate the entire set of instructions?

1. (assign val (op make-compiled-procedure) (label entry2) (reg env))
2. (goto (label after-lambda1))
3. entry2
4. (assign env (op compiled-procedure-env) (reg proc))
5. (assign env (op extend-environment) (const (x y)) (reg argl) (reg env))
6. (save continue)
7. (save env)
8. (assign proc (op lookup-variable-value) (const =) (reg env))
9. (assign val (const 0))
10. (assign argl (op list) (reg val))
11. (assign val (op lookup-variable-value) (const x) (reg env))
12. (assign argl (op cons) (reg val) (reg argl))
13. (test (op primitive-procedure?) (reg proc))
14. (branch (label primitive-branch14))
15. compiled-branch13
16. (assign continue (label after-call12))
17. (assign val (op compiled-procedure-entry) (reg proc))
18. (goto (reg val))
19. primitive-branch14
20. (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
21. after-call12
22. (restore env)
23. (restore continue)
24. (test (op false?) (reg val))
25. (branch (label false-branch4))
26. true-branch5
27. (assign val (op lookup-variable-value) (const y) (reg env))
28. (goto (reg continue))
29. false-branch4
30. (assign proc (op lookup-variable-value) (const bar) (reg env))
31. (save continue)
32. (save proc)
33. (assign val (op lookup-variable-value) (const y) (reg env))
34. (assign argl (op list) (reg val))
35. (save argl)
36. (assign proc (op lookup-variable-value) (const foo) (reg env))
37. (assign val (op lookup-variable-value) (const x) (reg env))
38. (assign argl (op list) (reg val))
39. (test (op primitive-procedure?) (reg proc))
40. (branch (label primitive-branch8))
41. compiled-branch7
42. (assign continue (label after-call6))
43. (assign val (op compiled-procedure-entry) (reg proc))
44. (goto (reg val))
45. primitive-branch8
46. (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
47. after-call6
48. (restore argl)
49. (assign argl (op cons) (reg val) (reg argl))
50. (restore proc)
51. (restore continue)
52. (test (op primitive-procedure?) (reg proc))
53. (branch (label primitive-branch11))
54. compiled-branch10
55. (assign val (op compiled-procedure-entry) (reg proc))

```
56. (goto (reg val))
57. primitive-branch11
58. (assign val (op apply-primitive-procedure) (reg proc) (reg arg1))
59. (goto (reg continue))
60. after-call9
61. after-if3
62. after-lambda1
63. (perform (op define-variable!) (const bar) (reg val) (reg env))
64. (assign val (const ok))
65. (goto (reg continue))
```

**Part 10 (16 points)**

Our evaluator supports the idea of defining a variable, and the idea of setting a variable to a new value. But suppose we want to get rid of all the bindings for a variable, to “undefine” it?

Remember that our evaluator represents an environment as a list of frames, and a frame as a `cons` pair of two lists: the first being a list of the variables, and the second being a list of the corresponding values.

The attached code for the evaluator provides a data abstraction for frames:

```
(define (enclosing-environment env) (cdr env))
(define (first-frame env) (car env))
(define the-empty-environment '())

(define (make-frame variables values) (cons variables values))
(define (frame-variables frame) (car frame))
(define (frame-values frame) (cdr frame))
```

Let’s suppose we add a new special form to our evaluator: `(undefine var-name)` and that the evaluator dispatches expressions of this form to `(undefine-variable var env)` where `var` is bound to the variable name, and `env` is the environment pointer. We are going to complete the idea of undefining a variable, by taking advantage of the list structure of frames.

To do this, we need a way of processing a pair of lists in synchrony, a template for which is shown below. This procedure should behave much like a `filter`, except that it applies to pairs of lists:

```
(define (double-filter proc pr)
  (define (aux new1 new2 old1 old2)
    (if (or (null? old1) (null? old2))
        INSERT1
        (if (proc (car old1) (car old2))
            INSERT2
            INSERT3)))
  (aux '() '() (car pr) (cdr pr)))
```

For example:

```
(double-filter
  (lambda (p q) (= p q))
  (cons '(1 2 3) '(1 9 3)))
;Value: ((1 3) 1 3)
```

In completing the procedure, you may assume that standard list procedures such as `reverse` are available.

**46. Provide the expression for INSERT1**

**47. Provide the expression for INSERT2**

**48. Provide the expression for INSERT3**

We can use this procedure to remove all the bindings of a variable in a frame.

```
(define (undefine-single-frame var frame)
  (double-filter
    INSERT4
    frame))
```

#### 49. Complete INSERT4

Of course, there could be bindings for the same variable in many frames, so we need to undefine the variable in all frames of an environment. We also need to be careful, since we want to preserve the top level list structure of an environment (as other parts of the code may point to it). Here is a template for completing this. `Env` is a list of frames, our goal is to have the procedure return a pointer to the same list structure, but where each frame pointer is replaced with a pointer to the new frame, in which the bindings of the variables have been removed.

```
(define (undefine-variable var env)
  (if (eq? env the-empty-environment)
      'done
      (begin
        INSERT5
        (undefine-variable var (enclosing-environment env)))))
```

#### 50. Complete INSERT5

**Part 11 (24 points)**

For each of the following procedures, indicate the order of growth of the procedure in using three different resources:  $T(n)$ ,  $S(n)$ , and  $H(n)$ . Here,  $T(n)$  is the time required, measured by the number of primitive operations performed.  $S(n)$  is the space required, measured by the maximum depth of the stack. And finally,  $H(n)$  is the space required, but now measured by the number of new `cons` cells explicitly allocated from the heap (ignoring `cons` cells allocated by the system to handle frames and environments). In each case,  $n$  is the number of elements in the input list. Select your answer from the following:

1.  $O(1)$
2.  $O(n)$
3.  $O(n^2)$
4.  $O(2^n)$
5.  $O(\log n)$
6. none of the above

Remember that

```
(define (append l1 l2)
  (if (null? l1)
      l2
      (cons (car l1) (append (cdr l1) l2))))
```

```
(define (length lst)
  (if (null? lst)
      0
      (+ 1 (length (cdr lst)))))
```

51.

```
(define (rev1 lst)
  (if (null? lst)
      '()
      (append (rev1 (cdr lst))
              (list (car lst)))))
```

52.

```
(define (rev2 lst)
  (define (aux old new)
    (if (null? old)
        new
        (aux (cdr old)
              (cons (car old) new))))
  (aux lst '()))
```

53.

```
(define (rev3 lst)
  (define (aux old new)
    (if (= (length old) 0)
        new
        (aux (cdr old)
              (cons (car old) new))))
  (aux lst '()))
```

**Part 12 (17 points)**

We want to create procedures that have some memory about how they have been used, in particular, we would like them to return the value of their application from the most recent evaluation. For example:

```
(define mine (make-proc (lambda (x) (* x x))))
```

```
(mine 3)
;Value: none
```

```
(mine 5)
;Value: 9
```

```
(mine 20)
;Value: 25
```

Thus, the first time one of these procedures is applied, it returns the symbol `none`, the second time it returns the value of its first application, and so on. For simplicity, we will deal only with procedures of one argument.

Here is a template for the procedure:

```
(define make-proc
  COMPLETE
  ))
```

**54: Provide the code for COMPLETE. (You can do this in 6 lines of code.)**

**Part 13 (5 points)**

**55. Write SIMPLE Scheme expressions <E1>, <E2> so that the following expressions return different values:**

```
(let ((x 0))
  (or <E1> <E2>)
  x)
```

```
(let ((x 0))
  ((lambda (v1 v2) (or v1 v2))
   <E1>
   <E2>)
  x)
```

```
; ;META-CIRCULAR EVALUATOR
```

```
1. (define (m-eval expr env)
2.   (cond ((self-evaluating? expr) expr)
3.         ((variable? expr) (lookup-variable-value expr env))
4.         ((quoted? expr) (text-of-quotation expr))
5.         ((assignment? expr) (eval-assignment expr env))
6.         ((definition? expr) (eval-definition expr env))
7.         ((if? expr) (eval-if expr env))
8.         ((lambda? expr) (make-procedure
9.                           (lambda-parameters expr)
10.                          (lambda-body expr)
11.                          env))
12.        ((begin? expr) (eval-sequence (begin-actions expr) env))
13.        ((cond? expr) (m-eval (cond->if expr) env))
14.        ((let? expr) (m-eval (let->combination expr) env))
15.        ((application? expr)
16.         (m-apply (m-eval (operator expr) env)
17.                   (list-of-values (operands expr) env)))
18.        (else (error "Unknown expression type -- M-EVAL" expr))))

19. (define (m-apply op arguments)
20.   (cond ((primitive-procedure? op)
21.         (apply-primitive-procedure op arguments))
22.         ((compound-procedure? op)
23.         (eval-sequence
24.          (procedure-body op)
25.          (extend-environment (procedure-parameters op)
26.                             arguments
27.                             (procedure-environment op))))
28.        (else (error "Unknown procedure type -- APPLY" op))))

29. (define (list-of-values rands env)
30.   (if (no-operands? rands)
31.       '()
32.       (cons (m-eval (first-operand rands) env)
33.             (list-of-values (rest-operands rands) env))))

34. (define (eval-if expr env)
35.   (if (m-eval (if-predicate expr) env)
36.       (m-eval (if-consequent expr) env)
37.       (m-eval (if-alternative expr) env)))

38. (define (eval-sequence exprs env)
39.   (cond ((last-expr? exprs) (m-eval (first-expr exprs) env))
40.         (else (m-eval (first-expr exprs) env)
41.               (eval-sequence (rest-exprs exprs) env))))

42. (define (eval-assignment expr env)
43.   (set-variable-value! (assignment-variable expr)
44.                         (m-eval (assignment-value expr) env)
45.                         env))
```

```

45.             env))

46. (define (eval-definition expr env)
47.   (define-variable! (definition-variable expr)
48.     (m-eval (definition-value expr) env)
49.   env))

;;COMPOUND PROCEDURE ADT

50. (define (make-procedure parameters body env)
51.   (list procedure-tag parameters body env))
52. (define procedure-tag (list 'procedure))
53. (define (compound-procedure? expr)
54.   (tagged-list? expr procedure-tag))
55. (define (procedure-parameters p) (list-ref p 1))
56. (define (procedure-body p) (list-ref p 2))
57. (define (procedure-environment p) (list-ref p 3))

;;ENVIRONMENTS

;; Implement environments as a list of frames; parent environment is
;; the cdr of the list. Each frame will be implemented as a list
;; of variables and a list of corresponding values.
58. (define (enclosing-environment env) (cdr env))
59. (define (first-frame env) (car env))
60. (define the-empty-environment '())

61. (define (make-frame variables values) (cons variables values))
62. (define (frame-variables frame) (car frame))
63. (define (frame-values frame) (cdr frame))
64. (define (add-binding-to-frame! var val frame)
65.   (set-car! frame (cons var (car frame))))
66.   (set-cdr! frame (cons val (cdr frame))))

67. (define (extend-environment vars vals base-env)
68.   (if (= (length vars) (length vals))
69.       (cons (make-frame vars vals) base-env)
70.       (if (< (length vars) (length vals))
71.           (error "Too many args supplied" vars vals)
72.           (error "Too few args supplied" vars vals))))

73. (define (lookup-variable-value var env)
74.   (define (env-loop env)
75.     (define (scan vars vals)
76.       (cond ((null? vars) (env-loop (enclosing-environment env)))
77.             ((eq? var (car vars)) (car vals))
78.             (else (scan (cdr vars) (cdr vals)))))
79.     (if (eq? env the-empty-environment)
80.         (error "Unbound variable -- LOOKUP" var)
81.         (let ((frame (first-frame env)))

```

```

82.         (scan (frame-variables frame) (frame-values frame))))))
83. (env-loop env)

84. (define (set-variable-value! var val env)
85.   (define (env-loop env)
86.     (define (scan vars vals)
87.       (cond ((null? vars)
88.              (env-loop (enclosing-environment env)))
89.             ((eq? var (car vars))
90.              (set-car! vals val)          ; Same as lookup except for this
91.              (else (scan (cdr vars) (cdr vals))))))
92.     (if (eq? env the-empty-environment)
93.         (error "Unbound variable -- SET!" var)
94.         (let ((frame (first-frame env)))
95.             (scan (frame-variables frame) (frame-values frame))))))
96. (env-loop env)

97. (define (define-variable! var val env)
98.   (let ((frame (first-frame env)))
99.     (define (scan vars vals)
100.      (cond ((null? vars) (add-binding-to-frame! var val frame))
101.            ((eq? var (car vars)) (set-car! vals val))
102.            (else (scan (cdr vars) (cdr vals))))))
103.     (scan (frame-variables frame)
104.           (frame-values frame)))

; ;THE INITIAL ENVIRONMENT

105. (define (setup-environment)
106.   (let ((initial-env (extend-environment (primitive-procedure-names)
107.                                         (primitive-procedure-objects)
108.                                         the-empty-environment)))
109.     (define-variable! 'true #t initial-env)
110.     (define-variable! 'false #f initial-env)
111.     initial-env))

112. (define primitive-procedures
113.   (list
114.     (list 'apply m-apply)          ;APPLY IS A SPECIAL PRIMITIVE
115.     (list 'pair?                  ;SO IS PAIR?
116.           (lambda (x)
117.             (if (pair? x)
118.                 (if (not (eq? (car x) primitive-tag))
119.                     (not (eq? (car x) procedure-tag))
120.                     #f) #f)))
121.     (list 'car car)
122.     (list 'cdr cdr)
123.     (list 'cons cons)
124.     (list 'null? null?)
125.     (list 'list list)
126.     (list 'length length)

```

```

127. (list 'list-ref list-ref)
128. (list 'eq? eq?)
129. (list 'caar caar)
130. (list 'cadr cadr)
131. (list 'cdar cdar)
132. (list 'cddr cddr)
133. (list 'set-car! set-car!)
134. (list 'set-cdr! set-cdr!)
135. (list 'boolean? boolean?)
136. (list 'not not)
137. (list 'number? number?)
138. (list 'string? string?)
139. (list 'symbol? symbol?)
140. (list '> >)
141. (list '< <)
142. (list '= =)
143. (list '+ +)
144. (list '- -)
145. (list '* *)
146. (list 'error error)
147. (list 'caadr caadr)
148. (list 'caddr caddr)
149. (list 'cadadr cadadr)
150. (list 'cddddr cddddr)
151. (list 'cadddr cadddr)
152. (list 'read read)
153. (list 'newline newline)
154. (list 'display display)
155. ;                               ; ... more primitives
156. ))

157. (define (primitive-procedure-names) (map car primitive-procedures))
158. (define primitive-tag (list 'primitive)) ;REVISED FROM LECTURE
159. (define (primitive-procedure-objects)
160.   (map (lambda (proc) (list primitive-tag (cadr proc))) primitive-procedures))
161. (define (apply-primitive-procedure proc args)
162.   (apply (primitive-implementation proc) args)) ;SCHEME'S APPLY, NOT M-APPLY

163. (define (primitive-implementation proc) (cadr proc))
164. (define (primitive-procedure? proc) (tagged-list? proc primitive-tag))

;(define the-global-environment (setup-environment))

;;THE READ-EVAL-PRINT LOOP

165. (define (driver-loop)
166.   (prompt-for-input input-prompt)
167.   (let ((input (read)))
168.     (let ((output (m-eval input the-global-environment)))
169.       (announce-output output-prompt)
170.       (display output)))

```

```
171. (driver-loop))

172. (define (prompt-for-input string)
173. (newline) (newline) (display string) (newline))
174. (define (announce-output string)
175. (newline) (display string) (newline))

176. (define input-prompt ";;; M-Eval2 input:")
177. (define output-prompt ";;; M-Eval2 value:")
;Eval2

;;SYNTAX
178. (define (tagged-list? expr tag)
179. (if (pair? expr) (eq? (car expr) tag) #f))

180. (define (self-evaluating? expr)
181. (cond ((number? expr) #t)
182. ((string? expr) #t)
183. (else (boolean? expr))))

184. (define (quoted? expr) (tagged-list? expr 'quote))
185. (define (text-of-quotation expr) (cadr expr))

186. (define (variable? expr) (symbol? expr))
187. (define (assignment? expr) (tagged-list? expr 'set!))
188. (define (assignment-variable expr) (cadr expr))
189. (define (assignment-value expr) (caddr expr))

190. (define (definition? expr) (tagged-list? expr 'define))
191. (define (definition-variable expr)
192. (if (symbol? (cadr expr))
193. (cadr expr)
194. (caadr expr)))
195. (define (definition-value expr)
196. (if (symbol? (cadr expr))
197. (caddr expr)
198. (make-lambda (cdadr expr) (cddr expr)))) ; formal params, body

199. (define (lambda? expr) (tagged-list? expr 'lambda))
200. (define (lambda-parameters lambda-expr) (cadr lambda-expr))
201. (define (lambda-body lambda-expr) (cddr lambda-expr))
202. (define (make-lambda parms body) (cons 'lambda (cons parms body)))

203. (define (if? expr) (tagged-list? expr 'if))
204. (define (if-predicate expr) (cadr expr))
205. (define (if-consequent expr) (caddr expr))
206. (define (if-alternative expr)
207. (if (not (null? (cddddr expr))) (cddddr expr) 'false))
208. (define (make-if pred conseq alt) (list 'if pred conseq alt))

209. (define (begin? expr) (tagged-list? expr 'begin))
```

```

210. (define (begin-actions begin-expr) (cdr begin-expr))
211. (define (last-expr? seq) (null? (cdr seq)))
212. (define (first-expr seq) (car seq))
213. (define (rest-exprs seq) (cdr seq))
214. (define (sequence->expr seq)
215.   (cond ((null? seq) (error "empty sequence")) ;REVISED
216.         ((last-expr? seq) (first-expr seq))
217.         (else (make-begin seq))))
218. (define (make-begin exprs) (cons 'begin exprs))

219. (define (cond? expr) (tagged-list? expr 'cond))
220. (define (cond-clauses expr) (cdr expr))
221. (define (clause-predicate clause) (car clause))
222. (define (clause-expr clause) (sequence->expr (cdr clause)))
223. (define (make-cond clauses)
224.   (cons 'cond clauses))

225. (define (cond->if expr)
226.   (let ((clauses (cond-clauses expr)))
227.     (if (null? clauses)
228.         ''unspecified
229.         (let ((pred1 (clause-predicate (car clauses)))
230.               (expr1 (clause-expr (car clauses))))
231.           (if (eq? pred1 'else)
232.               expr1
233.               (make-if pred1 expr1
234.                         (cond->if (make-cond (cdr clauses))))))))))

235. (define (let? expr) (tagged-list? expr 'let))
236. (define (let-bound-variables expr) (map car (cadr expr)))
237. (define (let-values expr) (map cadr (cadr expr)))
238. (define (let-body expr) (cddr expr)) ;DIFFERS FROM LECTURE
239. (define (make-let bindings body)
240.   (cons 'let (cons bindings body)))

241. (define (let->combination expr)
242.   (let ((names (let-bound-variables expr))
243.         (inits (let-values expr))
244.         (body (let-body expr)))
245.     (make-application ;DIFFERS FROM LECTURE
246.       (make-lambda names body)
247.       inits))

248. (define (application? expr) (pair? expr))
249. (define (operator app) (car app))
250. (define (operands app) (cdr app))
251. (define (no-operands? rands) (null? rands))
252. (define (first-operand rands) (car rands))
253. (define (rest-operands rands) (cdr rands))
254. (define (make-application rator rands)
255.   (cons rator rands))

```