

MASSACHUSETTS INSTITUTE OF TECHNOLOGY  
 Department of Electrical Engineering and Computer Science  
 6.001—Structure and Interpretation of Computer Programs  
 Fall 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 section numbers.

Section	Time	Location	Rec. Instructor	Tutors
1	10:00	26-328	Randy Davis	Nicole Love
2	11:00	26-328	Randy Davis	Alex Rolfe
3	11:00	8-302	Tony Eng	Todd Atkins
4	12:00	8-302	Tony Eng	
5	1:00	26-204	Peter Szolovits	Alex Andersson
6	2:00	26-204	Peter Szolovits	
7	10:00	36-155	Konrad Tollmar	Aaron Strauss
8	12:00	36-153	Konrad Tollmar	Adnan Dosani

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 10 AM on Thursday, December 19. 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 Room 606 of 400 Technology Square. 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 December 19 for students to look at (but not remove from) Jill Fekete's office, Room 606, 400 Technology Square.

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

Good luck!

**Part 1 (15 points):**

For each part, a sequence of expressions is given, which you may assume is typed into a fresh 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.

```
(lambda (x) x)
```

2.

```
(2 + 3)
```

3.

```
((lambda (* + -) (* + -)) + 3 5)
```

4.

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

5.

```
(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 1 (stream-map (lambda (x) (+ x 2)) st)))

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

6.

```
(define f
  (lambda (g)
    (lambda (n)
      (n g))))

((f (lambda (x) (* x x))) (lambda (x) x))
```

**Part 2: (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.

The form you are going to add is a `random-or` expression, which has the form:

```
(random-or <exp1> <exp2> ... <expn>)
```

where `<expi>` is some arbitrary Scheme expression. The behavior of a `random-or` is to select one of its arguments, at random, and evaluate it. If the value is `false`, then `random-or` selects a **different** expression, and continues. Once it selects an expression with a non-`false` value, it returns **that value**. If none of the expressions is non-`false`, `random-or` returns `false`.

Assume that the following clause has been added to the evaluator:

```
((random-or? expr)
 (eval-ror expr env))
```

**7. Write a procedure `select-random`, which selects a random element from its argument, which is assumed to be a list. You may assume that the procedure `random` exists, and that `(random n)` returns an integer between 0 and  $n - 1$  at random. You may also assume that `length` exists. Remember that `(list-ref lst n)` will return the  $n$ th element of `lst`.**

If we have selected an element from a list, we want to create a new list that contains all the elements of the original, except this selected element.

**8. Write a procedure `remove`, such that `(remove elt lst)` will return a list containing all the elements of the original, except `elt`. You may assume that the argument `lst` does not contain any duplicates.**

Now, here is the template for evaluation of a `random-or` expression:

```
(define (eval-ror expr env)
  (let ((choices (cdr expr)))
    (pickAndTest choices env)))
```

```
(define (pickAndTest choices env)
  (cond ((null? choices) #f)
        (else INSERT)))
```

**9. Provide the code for `INSERT`, using `select-random` and `remove`.**

**Part 3: (12 points)**

An alternative way of adding a new form to the meta-circular evaluator is to desugar the form into an existing form. Suppose we want to add the special form `pure-or` to our evaluator. `pure-or` evaluates each of its arguments in order, from left to right. As soon as it reaches an argument that evaluates to a non-false value, it returns the value of that expression. If it reaches the end of the arguments without finding a non-false value, it returns `false`. We can do this with a syntactic transformation that converts a `pure-or` expression into an equivalent `cond` expression, for example

```
(pure-or a b c)
```

should be converted into

```
(cond (a a)
      (b b)
      (c c)
      (else #f))
```

To accommodate this, we can add the following clause to our evaluator:

```
((pure-or? exp)
 (m-eval (pure-or->cond exp) env))
```

where you may assume that `pure-or?` is a predicate for identifying `pure-or` expressions.

Assume that the data abstraction for manipulating `pure-or` expressions exists, namely:

- `pure-or-clauses` returns the list of clauses in an `pure-or`,
- `first-clause` returns the first clause in a list,
- `rest-clauses` returns the list of all clauses except the first, and
- `no-clauses?` returns true if its argument contains no clauses.

Using these data abstractions, here is the template for the procedure we need:

```
(define (pure-or->cond exp)
  (cons 'cond (createCondClauseList (pure-or-clauses exp))))
```

**10. Complete the definition for `createCondClauseList`**

**11. Under what conditions will this version of `pure-or` differ in behavior from or implemented as a special form?**

**Part 4: (20 points)**

In this part we are going to add a new form to the Explicit Control Evaluator, in this case the special form `cond` (note that in the original Explicit Control Evaluator, we only used `if` expressions for branching). Remember the form for a `cond`, e.g.:

```
(cond ((> x y) (do-something) (do-something-else some-args))
      (test2 (things-to-do x y z))
      (else 'always-do-this-as-end))
```

A `cond` consists of a sequence of clauses, each of which begins with a **predicate expression**. The `cond` evaluates each predicate in turn. As soon as it finds one that is non-false, it then evaluates the remaining expressions in that clause (note that there could be more than one), returning the value of the last one. If it reaches an expression beginning with the special keyword `else` it then evaluates the remaining expressions in that clause, returning the value of the last one.

We will assume the following procedures exist as primitive operations:

- `clauses` when applied to a `cond` expression will return a list of all the clauses of the `cond`;
- `no-clauses?` when applied to a list of clauses will return true if there are no remaining clauses;
- `first-clause` when applied to a list of clauses will return the first clause;
- `rest-clauses` when applied to a list of clauses will return a list of all but the first clause;
- `predicate` when applied to a clause will return the first subexpression, which should be tested for a non-false value; and
- `consequents` when applied to a clause will return all the other expressions of a clause, which are to be evaluated if the predicate was non-false.

To `eval-dispatch`, we add:

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

where

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

Below is a template of the code at the label `ev-cond`, which you will be completing below. (The lines are numbered for convenience, but are not part of the register machine code.) You may find it convenient to remember that the contract for `ev-sequence` is that the continuation is at the top of stack.

```
ev-cond
1  (assign unev (op clauses) (reg exp))
   cond-loop
2  (test (op no-clauses?) (reg unev))
3  (branch (label none-left))
4  (assign exp (op first-clause) (reg unev))
5  (save unev)
6  (save exp)
7  (assign exp (op predicate) (reg exp))
8  (test (op eq?) (const else) (reg exp))
9  (branch (label else-case))
10 (save continue)
11 (assign continue (label after-test))
12 (goto (label eval-dispatch))
   after-test
13 INSERT1
14 INSERT2
15 (test (op true?) INSERT3)
16 (branch (label do-consequents))
17 (restore unev)
18 INSERT4
19 (goto (label cond-loop))
   do-consequents
20 INSERT5
21 (save continue)
22 (assign exp (op consequents) (reg exp))
23 (goto INSERT6)
   else-case
24 INSERT7
25 INSERT8
26 (save continue)
27 (assign exp (op consequents) (reg exp))
28 (goto INSERT9)
   none-left
29 (assign val (const #f))
30 (goto INSERT10)
```

- 12: What single register machine instruction should be used for INSERT2?
- 13: What single register machine instruction should be used for INSERT3?
- 14: What should be used to complete the single register machine instruction at INSERT4?
- 15: What single register machine instruction should be used for INSERT5?
- 16: What single register machine instruction should be used for INSERT6?
- 17: What should be used to complete the single register machine instruction at INSERT7?
- 18: What single register machine instruction should be used for INSERT8?
- 19: What single register machine instruction should be used for INSERT9?
- 20: What should be used to complete the single register machine instruction at INSERT10?
- 21: What should be used to complete the single register machine instruction at INSERT11?



**Part 6: (20 points)**

Streams are a useful programming technique when we want to decouple the order of actual evaluation within the machine from the apparent order of evaluation of a computation.

This can be particularly useful when trying to compute numerical approximations. For example, exponentiation is given by the following series:

$$e^x \approx \frac{1}{0!} + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

where  $0! = 1$ .

Thus, if we were to successively add up terms in this approximation, we would get better and better estimates:

$$e^x \approx \frac{1}{0!}$$

$$e^x \approx \frac{1}{0!} + \frac{x}{1!}$$

$$e^x \approx \frac{1}{0!} + \frac{x}{1!} + \frac{x^2}{2!}$$

$$e^x \approx \frac{1}{0!} + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!}$$

We are going to compute this approximation using streams. Assume that the following procedures are available:

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

```
(define (stream-op-2 proc str1 str2)
  (cons-stream (proc (stream-car str1) (stream-car str2))
               (stream-op-2 proc (stream-cdr str1) (stream-cdr str2))))
```

and that the following has been defined:

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

The following expression should create a stream of integers, starting at 1.

```
(define ints
  (cons-stream 1
    (stream-map ADD1 ADD2)))
```

For example, if we print out the first 5 elements of `ints`:

```
(stream-print ints 5)
```

```
1
2
3
4
5
```

**23. Provide the missing expression for `ADD1`.**

**24. Provide the missing expression for `ADD2`.**

Complete the following expression to create a stream of factorials, where  $0! = 1$ ,  $1! = 1$ , and  $n! = n \cdot (n - 1)!$ .

```
(define facts
  (cons-stream 1
    (stream-op-2 * ADD3 ADD4)))
```

For example:

```
(stream-print facts 5)
```

```
1
1
2
6
24
```

**25. Provide the missing expression for `ADD3`.**

**26. Provide the missing expression for `ADD4`.**

The following expression should generate a stream of powers of the argument `x`.

```
(define (powers x)
  (cons-stream 1
    (stream-map ADD5 ADD6)))
```

```
(stream-print (powers 2) 5)
```

```
1
2
4
8
16
```

**27. Provide the missing expression for ADD5.**

**28. Provide the missing expression for ADD6.**

We can then create

```
(define inv-facts (stream-map (lambda (x) (/ 1 x)) facts))
```

```
(define (terms x)
  (stream-op-2 * (powers x) inv-facts))
```

To complete the process, we need to successively add up terms to create approximations to the exponentiation of a power.

```
(define (approx x)
  (cons-stream 0
    (stream-op-2 + ADD7 ADD8)))
```

**29. Provide the missing expression for ADD7.**

**30. Provide the missing expression for ADD8.**

**Part 7. (24 points)**

Suppose you have a set of integers, represented as a list. For each of the following `insert` 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 already in the list `db`. 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

Also, for each procedure indicate whether, after some arbitrary number of insertions starting from an empty set, the result is ordered in:

- A: increasing order;
- B: decreasing order;
- C: some other arrangement.

31.

```
(define (insert elt db)
  (cond ((null? db) (list elt))
        ((< elt (car db))
         (cons elt db))
        (else (cons (car db)
                     (cons elt (cdr db))))))
```

32.

```
(define (insert elt db)
  (if (= (length db) 0)
      (list elt)
      (if (> elt (car db))
          (cons elt db)
          (cons (car db)
                (insert elt (cdr db)))))
```

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

33.

```
(define (insert elt db)
  (define (help cur elt next)
    (if (null? next)
        (set-cdr! cur (list elt))
        (if (< elt (car next))
            (let ((new (list elt)))
              (set-cdr! cur new)
              (set-cdr! new next))
            (help next elt (cdr next)))))
  (if (null? db)
      (list elt)
      (if (< elt (car db))
          (cons elt db)
          (begin
             (help db elt (cdr db))
             db))))
```

**Part 8. (12 points)**

We want to create a procedure called `repeated` that takes two arguments, a procedure `p` of type `number  $\mapsto$  number`, and a number `n`. `Repeated` returns a procedure of type `number  $\mapsto$  number` that will apply the procedure `P` `n` times in succession. For example, assuming we have a definition for `repeated`, we can use it to implement multiplication by successive additions, and exponentiation by successive multiplications:

```
(define (mul a b)
  ((repeated (lambda (x) (+ a x)) b) 0))

(define (my-expt c n)
  ((repeated (lambda (y) (mul y c)) n) 1))
```

**34. Provide a definition for `repeated`.**

**Part 9 (16 points):**

The following code fragment was generated using the Scheme compiler.

35. What Scheme expression was compiled to generate line 11?
36. What Scheme expression was compiled to generate lines 8–21?
37. What Scheme expression was compiled to generate lines 36–47?
38. What Scheme expression was compiled to generate lines 30–60?
39. What Scheme expression was compiled to generate lines 8–61?
40. What Scheme expression was compiled to generate lines 1–62?
41. 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 arg1) (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 arg1 (op list) (reg val))
11. (assign val (op lookup-variable-value) (const x) (reg env))
12. (assign arg1 (op cons) (reg val) (reg arg1))
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 arg1))
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 arg1 (op list) (reg val))
35. (save arg1)
36. (assign proc (op lookup-variable-value) (const foo) (reg env))
37. (assign val (op lookup-variable-value) (const x) (reg env))
38. (assign arg1 (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 arg1))
47. after-call6
48. (restore arg1)
49. (assign arg1 (op cons) (reg val) (reg arg1))
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 (18 points)**

In a mark-sweep garbage collector, the MARK phase recursively traces all structure accessible from the root. It marks each new cell it reaches, so that it can avoid retracing substructures. The SWEEP phase scans memory, finding all unmarked cells, which it strings together into a list. The pointer to that list is kept in the FREE register. When memory is needed (e.g., in CONS) the cell pointed at by FREE is allocated and FREE is set to the CDR of that cell. When the free list becomes empty, the garbage collector is called again. The MARK phase requires a stack to control its recursive operation. This stack cannot be part of the garbage-collected memory, so we allocate a special segment of memory for holding the garbage-collection stack. It is used to save a register with the PUSH instruction. The result is restored with the POP instruction. Here is a simple version of such a garbage collector.

```
garbage-collect
  (assign thing (reg root))
  (assign continue (label sweep))
mark
  (test (op pointer-to-pair?) (reg thing))
  (branch (label mark-pair))
mark-done
  (goto (reg continue))
mark-pair
  (assign mark-flag (op vector-ref) (reg the-marks) (reg thing))
  (test (op =) (reg mark-flag) (const 1))
  (branch (label mark-done))
  (perform (op vector-set!) (reg the-marks) (reg thing) (const 1))
  (push thing)
  (push continue)
  (assign continue (const mark-cdr))
  (assign thing (op vector-ref) (reg the-cars) (reg thing))
  (goto (label mark))
mark-cdr
  (pop continue)
  (pop thing)
  (assign thing (op vector-ref) (reg the-cdrs) (reg thing))
  (goto (label mark))
sweep
  (assign free (const the-empty-list))
  (assign scan (op -1+) (reg memtop))
sweep-loop
  (test (op negative?) (reg scan))
  (branch gc-done)
  (assign mark-flag (op vector-ref) (reg the-marks) (reg scan))
  (test (op =) (reg mark-flag) (const 1))
  (branch (label unmark))
  (perform (op vector-set!) (reg the-cdrs) (reg scan) (reg free))
  (assign free (reg scan))
  (assign scan (op -1+) (reg scan))
  (goto (label sweep-loop))
unmark
  (perform (op vector-set!) (reg the-marks) (reg scan) (const 0))
  (assign scan (op -1+) (reg scan))
  (goto (label sweep-loop))
gc-done
```

Let's examine the behavior of this mark-sweep garbage collector for a small example. Consider, for example, the following initial contents of a tiny memory of 8 pairs (drawn below), where the ROOT register contains the pointer P1.

Address	0	1	2	3	4	5	6	7
the-cars	P3	P5	N3	P0	P7	N1	N4	N2
the-cdrs	P2	P2	P4	P6	E0	P7	N2	E0
the-marks	0	0	0	0	0	0	0	0

**42:** Assume that the MEMTOP register contains a P8. We now start the machine at GARBAGE-COLLECT. Show the state of the list-structured memory when the control reaches the place named SWEEP. Please use the diagram in the answer sheet.

Address	0	1	2	3	4	5	6	7
the-cars								
the-cdrs								
the-marks								

**43:** Show the state of the memory when control reaches GC-DONE. Please use the diagram in the answer sheet.

Address	0	1	2	3	4	5	6	7
the-cars								
the-cdrs								
the-marks								

**44:** What is the contents of the FREE register when control reaches GC-DONE?

**45:** What is the maximum number of items that are on the garbage collector stack at any time in running this example? Explain your answer.

**46:** Assume that our computer has 64,000 pair cells rather than the 8 used in the illustration in the previous parts. Given that PUSH and POP are implemented with a finite auxiliary memory of 100 cells, describe (in a sentence of English) a data structure that will cause the stack to overflow (run out of memory).

**Part 11 (10 points)**

It is sometimes convenient to keep track of what arguments a procedure has been called on. The following procedure keeps track of this for any procedure of one numerical argument:

```
(define (history proc)
  (let ((calls '()))
    (lambda (arg)
      (cond ((number? arg)
             INSERT12)
            ((eq? arg 'calls)
             INSERT13)
            ((eq? arg 'reset)
             INSERT14)
            (else (error "illegal argument"))))))
```

For example:

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

(square 'calls)
;Value: #f

(square 5)
;Value: 25

(square 10)
;Value: 100

(square 'calls)
;Value: (10 5)

(square 'reset)
;Value: done

(square 'calls)
;Value: #f
```

**47.** What code should be provided in place of INSERT12, so that the above behavior occurs?

**48.** What code should be provided in place of INSERT13, so that the above behavior occurs (i.e. the list of supplied arguments is returned)?

**49.** What code should be provided in place of INSERT14, so that the above behavior occurs (i.e. the list of supplied arguments is reset to the empty list, and the symbol done is returned)?

**Part 12 (20 points)**

A tree is a list of lists, for example:

```
(define test '(1 (2 (3 (4 (5))) 6)))
;Value: "test --> (1 (2 (3 (4 (5))) 6))"
```

The following procedure provides a general method for manipulating trees:

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

For each of the following examples, select from the following set of choices for the returned value. Note that a choice may be used as an answer more than once.

Choices:

1. (1 2 3 4 5 6)
  2. (1 (2 (3 (4 (5))) 6)))
  3. (6 5 4 3 2 1)
  4. ((6 (((5) 4) 3) 2) 1)
  5. 6
  6. 21
  7. 0
  8. None of the above
50. (tree-ops test '() list cdr car append)
  51. (tree-ops test '() list car cdr append)
  52. (tree-ops test '() (lambda (x) x) cdr car (lambda (x y) (append x (list y))))
  53. (tree-ops test 0 (lambda (x) x) cdr car +)
  54. (tree-ops test 0 (lambda (x) x) car cdr max)

```

;;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))

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 'cdadr cdadr)
150. (list 'cdddr cdddr)
151. (list 'caddr caddr)
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? (caddr expr))) (caddr expr) 'false))
208. (define (make-if pred consequent alt) (list 'if pred consequent 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))
```