

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
 Department of Electrical Engineering and Computer Science
 6.001—Structure and Interpretation of Computer Programs
 Spring Semester, 2001

Final Exam

Closed Book – 3 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.

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	9:00	26-314	Trevor Darrell	Wes Beebee
2	10:00	34-310	Joel Moses	Christine Alvarado
3	10:00	26-314	Trevor Darrell	Damon Mosk-Aoyama
4	10:00	34-304	Berthold Horn	Jacob Strauss
5	11:00	34-301	Joel Moses	Christine Alvarado
				Damon Mosk-Aoyama
6	11:00	34-304	Berthold Horn	Jacob Strauss
7	12:00	34-301	Tony Eng	Nicole Love
8	1:00	34-303	Gill Pratt	Wes Beebee
9	1:00	34-301	Tony Eng	Chris Cheng
				Sourabh Niyogi
10	2:00	34-303	Gill Pratt	Nicole Love
				Ben Vandiver
11	11:00	26-314	Saman Amarasinghe	Sourabh Niyogi
12	12:00	26-314	Saman Amarasinghe	Chris Cheng

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 Tuesday, May 29. After that time, you may (1) wait until grades are mailed by the Registrar in June; (2) access your grades via WEBSIS (<http://student.mit.edu/>), or (3) contact the course secretary, Jill Fekete, in NE43-711. 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 29 for students to look at (but not remove from) Jill Fekete's office, Room NE43-711.

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

Good luck!

Part 1 (15 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.

1:

```
((lambda (x + y) (+ y x)) 5 * 3)
```

2:

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

```
((f 10) inc)
```

3:

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

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

4:

```
(define a (list 'a 'b))
```

```
(define b (list 'a a 'b))
```

```
(set-cdr! (cdr a) (caddr b))
```

b

5:

```
(let ((temp expt))
  (lambda (n)
    (lambda (x) (temp x n))))
```

Part 2 (50 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 a `until-not` expression, for example:

```
(until-not (< i limit)
  (process i)
  (set! i (inc i))
  i)
```

A `until-not` 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, the sequence

```
(process i)
(set! i (inc i))
i
```

. The evaluation of a `until-not` 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.
- Then the end test is evaluated.
- If the value of the end test is false, then the value returned for the entire `until-not` expression is the value computed evaluating the body in the previous step.
- If the value of the end test is true, then the body is evaluated again, and this continues until the end test is **false**.

In the example shown, the expression would ultimately return the value of `limit`, assuming that `i` was initially bound to some value no larger than `limit` before the `until-not` expression was evaluated.

We need a data abstraction for dealing with `until-not` expressions.

6: Define the predicate `until-not`?

7: Define the selector `until-not-test` which should return the end test of a correctly constructed `until-not` expression.

8: Define the selector `until-not-body` which should return the body of a correctly constructed `until-not` expression.

Assume that the procedure `eval-until-not` (which you will shortly complete) handles the actual evaluation of a `until-not` expression. Then the following clause can be added to `meval`:

```
((until-not? exp)
 (eval-until-not exp env))
```

Here is a partial implementation of `eval-until-not`.

```
(define (eval-until-not exp env)
  (let ((test (until-not-test exp))
        (body (until-not-body exp)))
    COMPLETE-THIS
  ))
```

9: Complete the procedure `eval-until-not` by providing the missing code for `COMPLETE-THIS`.

An alternative method for handling a new special form is to use syntactic conversion, that is, to reformulate the expression as another kind of expression, already handled by the evaluator, and then evaluate that new expression instead.

We can convert a `until-not` expression into a recursive loop. For example, if our `until-not` expression had the form

```
(until-not test exp1 exp2 exp3)
```

then we could in principle rewrite this as:

```
(let ()
  (define (loop)
    (let ((return-value !INSERT1!))
      (if (not test)
          !INSERT2!
          !INSERT3!)))
  (loop))
```

Note the use of the `let` expression to create a frame within which we can bind the name `loop` to a value for future reference.

10: What expression should be substituted for !INSERT1!

11: What expression should be substituted for !INSERT2!

12: What expression should be substituted for !INSERT3!

Now we are going to write a syntactic converter that will execute this conversion for any `until-not` expression. In other words, to `eval` we add a clause:

```
((until-not? expr)
 (eval (until-not->loop expr) env))
```

Complete the template below:

```
(define (until-not->loop exp)
  (let ((test (until-not-test exp))
        (body (until-not-body exp)))
    (list 'let
          '()

          !INSERT4!

          '(loop))))
```

13: What expression(s) should be supplied for !INSERT4!?

We can also add `until-not` expressions to the Explicit Control Evaluator. To `eval-dispatch`, we add:

```
(test (op until-not?) (reg exp))
(branch (label ev-until-not))
```

Below is a template of the code at the label `ev-until-not`, which you will be completing below.

```
ev-until-not
  (save continue)
  (save exp) ; save entire expression
  (assign unev (op until-not-test) (reg exp)) ; get end test
  (save unev) ; save for later
  (assign unev (op until-not-body) (reg exp)) ; get body
  (assign continue (label ev-after-until-not-body)) ; where to go when done
  (save env)
  (save continue)
  !INSERT5! ; evaluate the body
ev-after-until-not-body
  (restore env)
  !INSERT6!
  !INSERT7!
  (assign exp (reg unev)) ; get end test
  (assign continue (label ev-after-until-not-test))
  (save env)
  !INSERT8!
ev-after-until-not-test
  (restore env)
  (test (op true?) !INSERT9!)
  (branch (label until-not-loop))
  !INSERT10!
  !INSERT11!
  !INSERT12!
  !INSERT13!
until-not-loop
  !INSERT14!
  !INSERT15!
  !INSERT16!
  !INSERT17!
```

14: What code should be used for !INSERT5!

15: What code should be used for !INSERT6!

16: What code should be used for !INSERT7!

17: What code should be used for !INSERT8!

18: What code should be used for !INSERT9!

19: What code should be used for !INSERT10!

20: What code should be used for !INSERT11!

21: What code should be used for !INSERT12!

22: What code should be used for !INSERT13!

23: What code should be used for !INSERT14!

24: What code should be used for !INSERT15!

25: What code should be used for !INSERT16!

26: What code should be used for !INSERT17!

Part 4. (24 points)

Consider the five classes below – note the methods of each class and the inheritance structure. Note that one line of the PULL-ALL-NIGHTER method, called <EXPRESSION> is missing.

```
(define (make-athlete name)
  (lambda (msg)
    (case msg
      ((PUNT) (lambda (SELF) (display "let's play ball!") (newline)))
      ((NAME) (lambda (SELF) name))
      (else (no-method)))))

(define (make-poet name)
  (lambda (msg)
    (case msg
      ((PUNT) (lambda (SELF) (display "i have writer's block!") (newline)))
      ((NAME) (lambda (SELF) name))
      (else (no-method)))))

(define (make-engineer name)
  (lambda (msg)
    (case msg
      ((PUNT) (lambda (SELF) (display "i'll handwave!") (newline)))
      ((NAME) (lambda (SELF) name))
      (else (no-method)))))

(define (make-MIT-undergrad name)
  (let ((engineer-part (make-engineer name))
        (poet-part (make-poet name))
        (athlete-part (make-athlete name)))
    (define dispatch
      (lambda (msg)
        (case msg
          ((PULL-ALL-NIGHTER) (lambda (SELF)
                                (display "i'm so tired")
                                (newline)
                                <EXPRESSION>)) ; ;HERE IS WHAT WE WILL CHANGE!!
          ((PUNT) (lambda (SELF)
                    (display "there was a problem set due today?") (newline)))
          (else (find-method msg engineer-part athlete-part poet-part))))
      dispatch))

(define (make-6001-student name)
  (let ((undergrad-part (make-MIT-undergrad name)))
    (lambda (msg)
      (case msg
        ((PUNT) (lambda (SELF)
                  (display "i would never dream of punting 6.001")
                  (newline)
                  (ask self 'PULL-ALL-NIGHTER)))
        (else (get-method msg undergrad-part)))))
```

```

; Our familiar OOP system

(define (delegate to from message . args)
  (apply-method to from message args))

(define (apply-method in-object for-object message args)
  (let ((method (get-method message in-object)))
    (cond ((method? method)
           (apply method for-object args))
          ((eq? in-object for-object)
           (error "No method for" message 'in (ask in-object 'NAME)))
          (else (error "Can't delegate" message
                       "from" (ask for-object 'NAME)
                       "to" (ask in-object 'NAME))))))

;; THE FOLLOWING PROCEDURE IS A NEW ADDITION
(define (delegate-to-all to-list from message . args)
  ; assume that map applies the procedure in a left-to-right order
  (map (lambda (who) (apply-method who from message args))
       to-list))

(define (ask object message . args)
  (apply-method object object message args))

(define (get-method message object) ; single-inheritance
  (object message))

(define (find-method message . objects) ; multiple-inheritance
  (define (try objects)
    (if (null? objects)
        (no-method)
        (let ((method (get-method message (car objects))))
          (if (not (eq? method (no-method)))
              method
              (try (cdr objects))))))
  (try objects))

(define (method? x)
  (cond ((procedure? x) #T)
        ((eq? x (no-method)) #F)
        (else (error "Object returned this non-message:" x))))

(define no-method
  (let ((tag (list 'NO-METHOD)))
    (lambda () tag)))

```

In the following questions, we are going to replace <EXPRESSION> with a series of alternatives, and then evaluate

```
(define eric (make-6001-student 'eric))
```

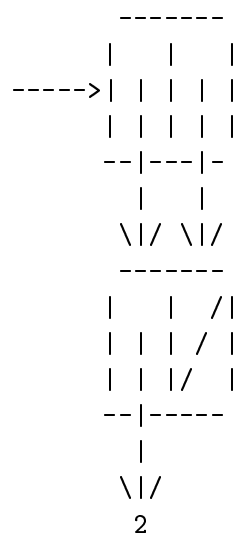
For each of the following possible Scheme expressions that could be used in place of <EXPRESSION> in the PULL-ALL-NIGHTER method, describe what happens when Eric is asked to PUNT, (`ask eric 'PUNT`). In particular, select from one of the possible outcomes listed on the next page (which you may tear out of the exam if it makes it easier for you), and numbered 1–12.

28. (`delegate engineer-part self 'PUNT`)
29. (`delegate self engineer-part 'PUNT`)
30. (`ask self 'PUNT`)
31. (`delegate self poet-part 'PULL-ALL-NIGHTER`)
32. (`delegate self dispatch 'PUNT`)
33. (`delegate self self 'PULL-ALL-NIGHTER`)
34. (`delegate poet-part athlete-part 'PUNT`)
35. (`delegate-to-all (list engineer-part poet-part athlete-part) self 'PUNT`)

- Outcome 1: i would never dream of punting 6.001
i'm so tired
let's play ball!
- Outcome 2: i would never dream of punting 6.001
i'm so tired
i have writer's block
- Outcome 3: i would never dream of punting 6.001
i'm so tired
i'll handwave!
- Outcome 4: i would never dream of punting 6.001
i'm so tired
i'll handwave!
i have writers' block
let's play ball
- Outcome 5: i would never dream of punting 6.001
i'm so tired
i'll handwave!
i would never dream of punting 6.001
i'm so tired
i have writers' block
i would never dream of punting 6.001
i'm so tired
let's play ball
- Outcome 6: i would never dream of punting 6.001
i'm so tired
there was a problem set due today?
- Outcome 7: i would never dream of punting 6.001
i'm so tired
i'm so tired
i'll handwave!
- Outcome 8: i would never dream of punting 6.001
i'm so tired
i'm so tired
;No method for pull-all-nighter in eric
;Type D to debug error, Q to quit back to REP loop:
- Outcome 9: i would never dream of punting 6.001
i'm so tired
i would never dream of punting 6.001
i'm so tired
there was a problem set due today?
- Outcome 10: i would never dream of punting 6.001
i'm so tired
i would never dream of punting 6.001
;No method for pull-all-nighter in eric
;Type D to debug error, Q to quit back to REP loop:
- Outcome 11: infinite loop
- Outcome 12: none of the above

Part 5 (20 points):

Consider the following box and pointer diagram:



36. What does this print as?

For each of the following expressions, indicate by selecting **true** or **false**, whether the expression gives rise to the box-and-pointer structure shown above.

37. `'((2) 2)`

38. `(list (list 2) 2)`

39. `(let ((temp (list 2)))
 (cons temp temp))`

40. `(let ((temp (list 2)))
 (let ((foo (cons (list 2) temp)))
 foo))`

41. `(let ((temp (list 2)))
 (let ((foo (cons (list 2) temp)))
 (set-car! foo temp)
 foo))`

42. `(let ((temp (list 2)))
 (let ((foo (cons (list 2) temp)))
 (set! (car foo) temp)
 foo))`

Part 6

Assume that `ran` is a primitive Scheme procedure that generates random numbers in the range 0 to 1, e. g.

```
(ran)
0.486726
```

```
(ran)
0.929204
```

```
(ran)
0.008849
```

```
(ran)
0.283186
```

Assume that successive calls to `RAN` *never* produce the same number.

Louis Reasoner wants to define a stream whose elements consist of different random numbers, as in the sequence above. He attempts to define a stream of random numbers as follows:

```
(define random-stream
  (cons-stream (ran)
               random-stream))
```

Lem E. Tweakit isn't sure that Louis' definition will work, and he suggests the following:

```
(define (make-random-stream)
  (cons-stream (ran)
               (make-random-stream)))

(define random-stream (make-random-stream))
```

The two friends show their work to Alyssa P. Hacker who suggests that they use `PRINT-STREAM` to examine the first few elements of their streams. Furthermore she suggests that they run their code on two different Scheme interpreters, one that implements lazy pairs using memoization, and one that does not.

Louis and Lem take her advice, and, just to be sure, they print their streams twice. Shown below are pairs of printouts, of the sort that either Louis or Lem might have produced.

Possible outcomes:

```
(print-stream random-stream)          ;;; OUTCOME A
0.486726 0.929204 0.008849 0.283186 ...
```

```
(print-stream random-stream)
0.486726 0.929204 0.008849 0.283186 ...
```

```
(print-stream random-stream)          ;;; OUTCOME B
0.486726 0.929204 0.008849 0.283186 ...
```

```
(print-stream random-stream)
0.486726 0.521080 0.297045 0.991644 ...
```

```
(print-stream random-stream)          ;;; OUTCOME C
0.486726 0.929204 0.008849 0.283186 ...
```

```
(print-stream random-stream)
0.365913 0.521080 0.297045 0.991644 ...
```

```
(print-stream random-stream)          ;;; OUTCOME D
0.486726 0.486726 0.486726 0.486726 ...
```

```
(print-stream random-stream)
0.486726 0.486726 0.486726 0.486726 ...
```

```
(print-stream random-stream)          ;;; OUTCOME E
0.486726 0.486726 0.486726 0.486726 ...
```

```
(print-stream random-stream)
0.591003 0.591003 0.591003 0.591003 ...
```

List all of the possible outcomes (chosen from A, B, C, D, E) that could have been produced in each of the following cases, or indicate **none** if none of these outcomes is possible.

43: Louis' definition; no memoization

44: Louis' definition; with memoization

45: Lem's definition; no memoization

46: Lem's definition; with memoization

Part 7 (10 points)

Write a procedure (`repeatedly-apply p n`), of type $(A \rightarrow A), \text{number} \rightarrow (A \rightarrow A)$, that returns a procedure that takes a single argument and applies `p` to it `n` times. So, for example,

```
((repeatedly-apply inc 10) 2)
;Value: 12
```

You may assume the following function has been defined

```
(define compose (lambda (f g) (lambda (x) (f (g x)))))
```

47: Using `compose`, write a definition for `repeatedly-apply` which works correctly for any value of `n` greater than or equal to 0.

Part 8 (20 points)

48. Write a procedure, `(accumulate-tree op id tree)` that accumulates the leaves of the tree using `op`, analogous to `accumulate` on lists.

So if `tree` has value

```
(( (1 2)
   3)
 (4
  (5 6))
 7)
```

then

```
(accumulate-tree + 0 tree)
;Value: 28
```

The second parameter, `id`, should be an identity element under the operation you give. That is, `op` applied to a value `v` and `id` should result in the value `v`. For example, `+` has the identity `0`, since adding `0` to anything results in that same thing.

Here is a template to complete. You can assume that `leaf?` is defined. **Hint:** use `map` to create a list of values, on which `accumulate` can operate.

```
(define accumulate-tree
  (lambda (op id tree)
    (if (leaf? tree)
        tree
        (accumulate
         INSERT-HERE
         )))
```

Supply the missing code for `INSERT-HERE`.

49. Write a procedure `(flatten-tree tree)`, that returns a flat list of the leaves of a tree.

For example,

```
(flatten-tree '((1) (2 3 4) ((5))))
```

should return `(1 2 3 4 5)`. It should call `accumulate-tree`, and should not call itself.

Here is a template to complete. You can assume that `leaf?` is defined.

```
(define flatten-tree
  (lambda (tree)
    (accumulate-tree INSERT-HERE )))
```

Supply the missing code for `INSERT-HERE`.

Part 9 (15 points)

For each of the following, indicate if the statement is **true** or **false**. Grading on this part will be 2 points for each correct answer, -2 points for each incorrect answer, and 0 points for no answer.

50: The halting theorem states that it is not possible to create a procedure that, when applied to any other procedure p can always determine if p will terminate with an answer.

51: There are some procedures that can be written in Scheme, but cannot be written in other programming languages, such as C, C++ or Java.

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

53: While the Mark-Sweep garbage collection algorithm can reclaim memory cells used by circular data structures that are not reachable from any “root” of good memory, the Stop-and-Copy garbage collection algorithm cannot.

54: It is not possible, without using mutation or print expressions, to write a procedure whose evaluation will enable us to determine if the evaluator is using normal order or applicative order evaluation.

Part 10 (20 points)

Consider the following five procedures that are expected to sort a list of numbers and return a new list whose elements are in increasing order. For each procedure, choose the best answer from the following choices for describing its behavior. Choices may be used more than once.

Choices:

1. Procedure correctly sorts a list.
2. Procedure will fail on lists of length 1.
3. Procedure may enter an infinite loop.
4. Procedure produces some other error.
5. Procedure produces a sorted list as output, but some of the elements of the input list may be missing.
6. Procedure only guarantees that the largest element is at the end of the list.
7. Procedure only guarantees that the smallest element is at the beginning of the list.

55:

```
(define (f1 lst)
  (cond ((null? lst) '())
        (else (let ((test (car lst)))
                  (let ((up (filter (lambda (x) (> x test)) (cdr lst)))
                        (down (filter (lambda (x) (<= x test)) (cdr lst))))
                    (append (f1 down)
                            (cons test (f1 up))))))))))
```

56:

```
(define (f2 lst)
  (cond ((null? lst) '())
        ((null? (cdr lst)) lst)
        (> (car lst) (cadr lst))
        (cons (cadr lst) (f2 (cons (car lst) (cddr lst))))
        (else (cons (car lst) (f2 (cdr lst))))))
```

57:

```
(define (f3 lst)
  (let ((lng (length lst)))
    (define (aux ls n)
      (if (= n 0)
          ls
          (aux (f3 ls) (- n 1))))
    (aux lst lng)))
```

58:

```
(define (f4 lst)
  (cond ((null? lst) '())
        ((null? (cdr lst)) lst)
        ((> (car lst) (cadr lst))
         (f4 (cdr lst)))
        (else (cons (car lst) (f4 (cdr lst))))))
```

59:

```
(define (f5 lst)
  (define (bubble ptr)
    (cond ((null? ptr) lst)
          ((null? (cdr ptr)) lst)
          ((> (car ptr) (cadr ptr))
           (let ((temp (car ptr)))
              (set-car! ptr (cadr ptr))
              (set-car! (cdr ptr) temp))
            (bubble lst))
          (else (bubble (cdr ptr)))))
  (bubble lst))
```



```

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 'cddddr cddddr)
151. (list 'caddddr caddddr)
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:")

;;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) (caddr 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) (caddr 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 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) (caddr 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))

```