

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
 Department of Electrical Engineering and Computer Science
 6.001—Structure and Interpretation of Computer Programs
 Fall Semester, 2003

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 the answer sheet 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	Joel Moses	Bryan Russell
2	10:00	36-155	Regina Barzilay	Ben Vandiver
3	11:00	26-328	Joel Moses	Dave Feinberg
4	12:00	36-144	Regina Barzilay	
5	1:00	26-204	Randy Davis	Limor Fried
6	2:00	26-204	Randy Davis	Vivienne Lee

No grades for 6.001 will be available until after 1 PM on Friday, 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 NE46-606. 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 NE46-606 (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):

Each question should be treated as independent. For each, the given sequence of expressions is evaluated in the order shown. Write the value that will be returned for the **last expression** in each sequence. If the sequence results in an 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.

Question 1:

```
(define (f x n)
  (if (= n 0)
      (list 'x)
      (cons x (f x (- n 1)))))
```

```
(f 1 3)
```

Question 2:

```
((lambda (f) (lambda (x y) (f y x))) -)
```

Question 3:

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

```
((f 5) (lambda (x) (+ x 1)))
```

Question 4:

```
(define (add-streams s1 s2)
  (cons-stream (+ (stream-car s1) (stream-car s2))
               (add-streams (stream-cdr s1) (stream-cdr s2))))
```

```
(define things (cons-stream 1 (add-streams things things)))
```

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

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

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

Question 5:

```
(define (repeated f n)
  (if (= n 0)
      (lambda (x) x)
      (lambda (x) (f ((repeated f (- n 1)) x)))))
```

```
((repeated (lambda (x) (expt x 2)) 3) 2)
```

Part 2 (29 points):

We are going to add a new special form to the Meta-Circular evaluator (the code for which is included at the end of the exam). This form is a `loop` expression, examples of which are:

```
(loop (i 1 inc) (= i 4)
      (newline)
      (display (list i (fact i))))
```

```
(1 1)
(2 2)
(3 6)
(4 24)
;Value: done
```

```
(define start-list '(1 3 5))
```

```
(loop (lst start-list cdr) (null? lst)
      (newline)
      (display (fact (car lst))))
```

```
1
6
120
;Value: done
```

The syntax of a `loop` is as follows. The first clause includes a **loop variable** (`i` in the first example), an expression whose value is **initial value** of the variable (`1` in the first example), and an **increment procedure** to apply to the loop variable on each iteration to create a new value for the loop variable (the value associated with `inc` in the first example). The next clause is an **end test**, an expression that will evaluate to `true` or `false`. The remaining expressions are the body of the loop.

The semantics of a `loop` is as follows. The loop variable is initially set to the value of its initialization expression. The end test is then evaluated. If the value is `true`, the loop exits, and the symbol `done` is returned. If not, the expressions in the body of the loop are evaluated. The incrementation procedure is then applied to the loop variable, and that variable is then bound to the returned value. The process then repeats.

Each of the following procedures extracts elements of a `loop`. Complete these definitions (assume that each would be applied to a full `loop` expression).

Question 6:

```
(define (loop-variable exp) YOUR-ANSWER)
```

Question 7:

```
(define (loop-initial-value exp) YOUR-ANSWER)
```

Question 8:

```
(define (loop-increment exp) YOUR-ANSWER)
```

Question 9:

```
(define (loop-end-test exp) YOUR-ANSWER)
```

Question 10:

```
(define (loop-body exp) YOUR-ANSWER)
```

To implement the special form, we add a dispatch clause to `m-eval`, and create a new evaluation procedure:

```
(define (m-eval exp env)
  (cond ...
    ((loop? exp)
     (eval-loop exp env))
    ...
    ((application? exp) ...)
    (else ...)))

(define (eval-loop exp env)
  (eval-loop-doit (loop-variable exp)
                 (loop-initial-value exp)
                 (loop-increment exp)
                 (loop-end-test exp)
                 (loop-body exp)
                 env))

(define (eval-loop-doit var init next end bod env)
  (let ((new-env (extend-environment
                 ANSWER-11
                 ANSWER-12
                 env)))
    (if ANSWER-13 ;test to see if done
        ANSWER-14 ;value to return
        (begin ANSWER-15 ; evaluate body
               ANSWER-16)))) ; go to next iteration
```

Question 11: Provide an expression for ANSWER-11. (Together with Question 12, this should create a new environment with the loop variable bound to a new value.)

Question 12: Provide an expression for ANSWER-12.

Question 13: Provide an expression for ANSWER-13 to determine if the loop has satisfied the end condition.

Question 14: Provide an expression ANSWER-14 to return the correct value from the loop.

Question 15: Provide an expression for ANSWER-15 to evaluate the body of the loop.

Question 16: Provide an expression for ANSWER-16 to handle the next loop iteration.

Part 3 (20 points):

Question 17: Write a recursive procedure (that is, a procedure with at least a linear growth in the number of deferred operations) to reverse a list.

Question 18: Write an iterative procedure (that is, a procedure with no deferred operations) to reverse a list.

Question 19: Write a recursive procedure to deep reverse a list, that is, to reverse each sublist as well, e.g.

```
(deep-reverse '(1 (2 3 (4 5)) 6))  
;Value: (6 ((5 4) 3 2) 1)
```

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

Here is a template:

```
(define (deep-reverse lst)  
  (cond ((null? lst) '())  
        ((not (pair? lst)) lst)  
        (else QUESTION-19)))
```

Provide the missing expression. You may use `reverse` (and of course any built-in Scheme function) in your answer.

Part 4 (16 points)

When we examined the Meta-Circular evaluator, we saw that one way to add new forms to our language was to create syntactic transformations, in which a special form was converted, using manipulation of the syntactic representation (e.g. the list structure of the expression), into a form that the evaluator could already handle. For example, a `cond` can be transformed into a nested set of `if` expressions.

Consider a new special form, called an `until`, such as `(until (= i 10) (set! i (+ i 1)) (display (list i (factorial i))))`. The format of an `until` is as follows: The first expression after the `until` tag is a test expression. When evaluating the `until` one first evaluates this expression. If it is `true`, the evaluation stops, and returns the symbol `done`. If not, then each subsequent expression is evaluated, and the entire `until` expression is evaluated again. This process continues until the test expression returns a true value.

Suppose we add the following clause to `m-eval`:

```
((until? exp) (m-eval (until->if exp) env))
```

where

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

Suppose we define

```
(define until-text cadr)
(define until-clauses caddr)
```

as selectors for the parts of an `until` expression.

Your job is to create the syntactic transformation `until->if`, which should take an `until` expression as input, and return a new expression, based on an `if`, whose evaluation will correctly implement the process described above. Note that this is not just a simple macro-like expansion, but needs to capture the recursive nature of the `until` process.

Question 20: Write the procedure `until->if`

Part 5 (20 points):

Suppose we want to add some simple type-checking to our language, that is, to specify conditions or constraints on the types of arguments that a procedure may take, with the idea that before we apply the procedure, we ensure that the supplied arguments meet those constraints. For example, we could extend our syntax to allow:

```
(define (foo (number? x) (list? y) z)
  some-body)
```

The idea is that when we are about to apply `foo` to some arguments, we will ensure that the first argument satisfies `number?` and the second argument satisfies `list?` before proceeding. Since there are no constraints specified on the last argument, anything is acceptable.

We will do this by making two changes to our evaluator. First, when creating a `lambda`, we will get the actual procedure object associated with each type checking clause (e.g., the procedure object for `number?`). Second, when we get to `m-apply` we will actually use those procedure objects to ensure that the arguments meets the constraints. We change the dispatch clause in `m-eval` to:

```
((lambda? exp)
 (make-procedure (CONVERT (lambda-parameters exp) env)
                 (lambda-body exp)
                 env))
```

Here is the framework for `convert`, which given a list of parameters (each of which is either a name, or a list of a procedure name and a variable name), should return a list of the same form, in which each name is kept, but the procedure name is replaced by the actual procedure. Thus `((number? x) (list? y) z)` would be converted to `((<actual procedure bound to number?> x) (<actual procedure bound to list?> y) z)`:

```
(define (convert params env)
  (cond ((null? params)
        '())
        ((symbol? (car params))
         (cons (car params) (convert (cdr params) env)))
        (else (cons QUESTION-21
                     (convert (cdr params) env))))))
```

Question 21: What expression should be provided for QUESTION-21?

Next, we change the standard version of `m-apply` (which is attached at the end of the exam for reference) to handle the type checking as follows:

```
(define (m-apply procedure arguments env)    ;;; added env
  (cond ((primitive-procedure? procedure)
        (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (let ((params (procedure-parameters procedure)))    ;;; ADDED
           (if (type-ok? params arguments)                  ;;; ADDED
               (eval-sequence
                (procedure-body procedure)
                (extend-environment
                 (map (lambda (x) (if (symbol? x) x (cadr x))) ;;; ADDED
                      params)
                 arguments
                 (procedure-environment procedure)))
               (error "incorrect argument type" procedure)))) ;;; ADDED
        (else (error "Unknown procedure type -- APPLY" procedure))))
```

To complete this change, we need to implement `type-ok?` which should check that each argument meets the specified type constraint:

```
(define (type-ok? params args)
  (cond ((null? params)
        QUESTION-22)
        ((null? args) #f)
        ((symbol? (car params))
         QUESTION-23)
        (QUESTION-24
         (type-ok? (cdr params) (cdr args)))
        (else #f)))
```

Question 22: What expression should be used for QUESTION-22?

Question 23: What expression should be used for QUESTION-23?

Question 24: What expression should be used for QUESTION-24?

Part 6 (18 points)

Each of the following three procedures will sort a list of numbers. For each, indicate the order of growth of the procedure:

- in time, as measured by the number of primitive operations to be performed, as a function of the length of the input list (call this $T(n)$);
- in space on the stack, as measured by the number of deferred operations that are stored on the stack during evaluation, as a function of the length of the input list (call this $S(n)$);
- in space in the heap, as measured by the number of `cons` cells used during the evaluation, as a function of the length of the input list (call this $H(n)$).

Question 25:

```
(define (max&rest lst)
  (define (help best seen rest)
    (if (null? rest)
        (cons best seen)
        (if (> best (car rest))
            (help best (cons (car rest) seen) (cdr rest))
            (help (car rest) (cons best seen) (cdr rest)))))
  (if (null? lst)
      '()
      (let ((next (help (car lst) '() (cdr lst))))
        (cons (car next)
              (max&rest (cdr next))))))
```

Question 26:

```
(define (max&remove lst)
  (if (null? lst)
      '()
      (let ((best (find-max lst))
            (rest (remove-first best lst)))
        (cons best (max&remove rest)))))

(define (find-max lst)
  (if (null? (cdr lst))
      (car lst)
      (max (car lst) (find-max (cdr lst)))))

(define (remove-first best rest)
  (if (null? rest)
      '()
      (if (= best (car rest))
          (cdr rest)
          (cons (car rest) (remove-first best (cdr rest))))))
```

Question 27:

```
(define (max&remove-new lst)
  (if (null? lst)
      '()
      (let ((best (find-max lst))
            (rest (remove-first-new best lst)))
          (cons best (max&remove-new rest)))))

(define (find-max lst)
  (if (null? (cdr lst))
      (car lst)
      (max (car lst) (find-max (cdr lst)))))

(define (remove-first-new best rest)
  (if (= best (car rest))
      (cdr rest)
      (begin (shift best rest (cdr rest))
              rest)))

(define (shift best where previous)
  (if (= best (car where))
      (set-cdr! previous (cdr where))
      (shift best (cdr where) where)))
```

Part 7 (16 points)

You are to write register machine code to find the maximum element in a list of numbers, using only the operations `>`, `car`, `cdr`, `null?`. First, write register machine code for an iterative process. Assume a non-empty list of numbers is in the register `arg`, that the `continue` register holds the continuation for the process, and that the result is returned in the register `val`. Use the register `temp` to hold any temporary values. Assume that `max` can clobber the contents of `arg` and that any other procedure that uses `max` as a subprocedure will first save the contents of `arg`.

Here is a template for the code:

```
max
  (assign val (op car) (reg arg))
max-loop
  (assign arg (op cdr) (reg arg))
  (test (op null?) (reg arg))
  (branch (label end-max))
  TO-BE-FILLED-IN
end-max
  (goto (reg continue))
```

Question 28:

Write code for `TO-BE-FILLED-IN` (this can be done in 7 lines, including labels).

Write register machine code that computes the same function, but using a recursive process. For clarity, here is a Scheme procedure that would accomplish this:

```
(define (max arg)
  (if (null? (cdr arg))
      (car arg)
      (let ((best (max (cdr arg))))
        (if (> (car arg) best)
            (car arg)
            best))))
```

Here is a template for this register machine process:

```
max
  (assign val (op car) (reg arg))
  (assign arg (op cdr) (reg arg))
  (test (op null?) (reg arg))
  (branch (reg continue))
  (save val)
  (save continue)
  (assign continue (label best-rest))
  (goto (label max))
best-rest
  TO-BE-FILLED-IN
```

Question 29:

Write register machine code for `TO-BE-FILLED-IN` (this can be done in 9 lines).

Part 8 (30 points)

For each of the following, indicate if the statement is **true** or **false**. Grading on this part will be 3 points for each correct answer, -2 points for each incorrect answer, and 0 points for no answer.

Question 30:

Consider all procedures having exactly one argument. The halting theorem states that it is not possible to create a procedure that, when applied to any other procedure p and any argument x can always determine if p will terminate with an answer.

Question 31:

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

Question 32: In dynamic scoping, free variables in a procedure body are referenced in the caller's environment, rather than in the environment in which the procedure was defined.

Question 33:

The Mark-Sweep garbage collection algorithm cannot reclaim memory cells used by circular data structures that are not reachable from any "root" of good memory.

Question 34:

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.

Question 35:

Any tree can be represented as a graph.

Question 36:

A stack is a kind of **FIFO** (or first in, first out) data structure, because the first element stored in the stack is also the first element to be removed.

Question 37:

A hash table implements data retrieval by using a hash function to determine which of many association lists in which to search for information associated with some lookup key, while an "a-list" or association list simply looks up data associated with a lookup key in a single list.

Question 38:

Serialization guarantees that shared variables in concurrent processes will always have a unique value at the end of the concurrent processing.

Question 39:

A lookup operation in an association list must have a time order of growth that is at least linear in the length of the list.

Part 9. (18 points)

Consider the environment diagram that results from evaluating the following expressions in this order. We suggest you draw your own environment diagram in the space below, in order to answer the following questions. Indicate the global environment by GE, and then be sure to label any additional environments, E1, E2, etc, **in the order in which they are created**.

```
(define (snoc x y) (lambda (m) (m x y)))
(define (rac z) (z (lambda (p q) p)))
(define x (snoc (quote a) (quote b)))
(define result (rac x))
```



Find the following procedure objects in your diagram and give them these labels (i.e. p1, p2, p3 or p4):

- P1 is the double bubble with parameters `p`, `q` and body `p`.
- P2 is the double bubble with parameters `x`, `y` and body `(lambda (m) (m x y))`.
- P3 is the double bubble with parameters `m` and body `(m x y)`.
- P4 is the double bubble with parameters `z` and body `(z (lambda (p q) p))`.

For each of the following questions, answer with a symbol, or a procedure object label (P1, P2, P3, P4) corresponding to your environment diagram.

The global environment has bindings for `snoc`, `rac`, `x` and `result`.

Question 40: What is the value bound to `snoc`?

Question 41: What is the value bound to `rac`?

Question 42: What is the value bound to `x`?

Question 43: What is the value bound to `result`?

The first frame of environment E1 has bindings for `x` and `y`.

Question 44: What is the value bound to `x`?

Question 45: What is the value bound to `y`?

The first frame of environment E2 has a binding for z.

Question 46: What is the value bound to z?

The first frame of environment E3 has a binding for m.

Question 47: What is the value bound to m?

The first frame of environment E4 has bindings for p and q.

Question 48: What is the value bound to p?

Question 49: What is the value bound to q?

Indicate the enclosing environment for each procedure argument:

Question 50: p1:

Question 51: p2:

Question 52: p3:

Question 53: p4:

Indicate the enclosing environment for each environment:

Question 54: E1:

Question 55: E2:

Question 56: E3:

Question 57: E4:

Part 10. (16 points)

Consider the classes below – note the methods of each class and the inheritance structure. Note that one line of the PUNT method for an engineer, will be changed in the questions.

```
(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)
  (let ((athlete-part (make-athlete name)))
    (lambda (msg)
      (case msg
        ((PUNT) (lambda (SELF) (display "i have writer's block!") (newline)
                               (ask athlete-part 'punt)))
        ((NAME) (lambda (SELF) (display "does anyone really have a name?"
                                         (newline) 'all-names-are-ephemeral))
                 (else (get-method msg athlete-part)))))))

(define (make-engineer name)
  (let ((poet-part (make-poet name))
        (athlete-part (make-athlete name)))
    (lambda (msg)
      (case msg
        ((PUNT) (lambda (SELF)
                  (display "I'm so tired!") (newline)
                  (ask self 'punt)) ;; HERE IS WHERE WE WILL CHANGE EXPRESSIONS
                 (else (find-method msg athlete-part poet-part)
                        ;; AND HERE IS WHERE WE WILL CONSIDER CHANGING ORDER
                        )))))
    ))))
```

```
; 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))))))

(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 consider some variations on the definition of an engineer.

Question 58:

First, suppose we evaluate

```
(define eric (make-engineer 'eric))  
  
(ask eric 'PUNT)
```

What happens?

Question 59: Now suppose we replace the last line of the PUNT method for engineers with

```
(ask poet-part 'PUNT)
```

If we (ask eric 'PUNT), what gets printed out on the monitor?

Question 60: Suppose instead that we replace the last line of the PUNT method with

```
(ask athlete-part 'PUNT)
```

If we (ask eric 'PUNT), what gets printed out on the monitor?

Question 61: Suppose instead that we replace the last line of the PUNT method with

```
(delegate self poet-part 'punt)
```

If we (ask eric 'PUNT), what gets printed out on the monitor?

Question 62: Suppose we go back to the original definition of an engineer. Now, if we (ask eric 'NAME), what gets printed out on the monitor?

Question 63: Finally, suppose we replace the last line of an engineer class with

```
(find-method msg poet-part athlete-part)
```

and again we (ask eric 'name). What gets printed out on the monitor?

The Core Evaluator

```

(define (m-eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp) (make-procedure (lambda-parameters exp) (lambda-body exp) env))
        ((begin? exp) (eval-sequence (begin-actions exp) env))
        ((cond? exp) (m-eval (cond->if exp) env))
        ((application? exp)
         (m-apply (m-eval (operator exp) env) (list-of-values (operands exp) env)))
        (else (error "Unknown expression type -- EVAL" exp))))

(define (m-apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment (procedure-parameters procedure)
                             arguments
                             (procedure-environment procedure))))
        (else (error "Unknown procedure type -- APPLY" procedure))))

(define (list-of-values exps env)
  (cond ((no-operands? exps) '())
        (else (cons (m-eval (first-operand exps) env)
                      (list-of-values (rest-operands exps) env)))))

(define (eval-if exp env)
  (if (m-eval (if-predicate exp) env)
      (m-eval (if-consequent exp) env)
      (m-eval (if-alternative exp) env)))

(define (eval-sequence exps env)
  (cond ((last-exp? exps) (m-eval (first-exp exps) env))
        (else (m-eval (first-exp exps) env)
                (eval-sequence (rest-exps exps) env))))

(define (eval-assignment exp env)
  (set-variable-value! (assignment-variable exp)
                        (m-eval (assignment-value exp) env)
                        env))

(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
                    (m-eval (definition-value exp) env)
                    env))

```

Representing Expressions

```

(define (tagged-list? exp tag)
  (and (pair? exp) (eq? (car exp) tag)))

(define (self-evaluating? exp)
  (or (number? exp) (string? exp) (boolean? exp)))

(define (quoted? exp) (tagged-list? exp 'quote))
(define (text-of-quotation exp) (cadr exp))

(define (variable? exp) (symbol? exp))
(define (assignment? exp) (tagged-list? exp 'set!))
(define (assignment-variable exp) (cadr exp))
(define (assignment-value exp) (caddr exp))

(define (definition? exp) (tagged-list? exp 'define))
(define (definition-variable exp)
  (if (symbol? (cadr exp)) (cadr exp) (caadr exp)))
(define (definition-value exp)
  (if (symbol? (cadr exp))
      (caddr exp)
      (make-lambda (cdadr exp) (caddr exp)))) ; formal params, body

(define (lambda? exp) (tagged-list? exp 'lambda))
(define (lambda-parameters lambda-exp) (cadr lambda-exp))
(define (lambda-body lambda-exp) (caddr lambda-exp))
(define (make-lambda parms body) (cons 'lambda (cons parms body)))

(define (if? exp) (tagged-list? exp 'if))
(define (if-predicate exp) (cadr exp))
(define (if-consequent exp) (caddr exp))
(define (if-alternative exp)
  (if (not (null? (caddr exp))) (caddr exp) 'false))
(define (make-if pred consequent alt) (list 'if pred consequent alt))

(define (cond? exp) (tagged-list? exp 'cond))
(define (cond-clauses exp) (cdr exp))

(define (begin? exp) (tagged-list? exp 'begin))
(define (begin-actions begin-exp) (cdr begin-exp))
(define (last-exp? seq) (null? (cdr seq)))
(define (first-exp seq) (car seq))
(define (rest-exps seq) (cdr seq))
(define (sequence->exp seq)
  (cond ((null? seq) seq)
        ((last-exp? seq) (first-exp seq))
        (else (make-begin seq))))
(define (make-begin exp) (cons 'begin exp))

(define (application? exp) (pair? exp))

```

```
(define (operator app) (car app))
(define (operands app) (cdr app))
(define (no-operands? args) (null? args))
(define (first-operand args) (car args))
(define (rest-operands args) (cdr args))
```

Representing procedures

```
(define (make-procedure parameters body env)
  (list 'procedure parameters body env))
(define (compound-procedure? exp)
  (tagged-list? exp 'procedure))
(define (procedure-parameters p) (list-ref p 1))
(define (procedure-body p) (list-ref p 2))
(define (procedure-environment p) (list-ref p 3))
```

Representing 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.
```

```
(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))
(define (add-binding-to-frame! var val frame)
  (set-car! frame (cons var (car frame)))
  (set-cdr! frame (cons val (cdr frame))))

(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
          (error "Too many args supplied" vars vals)
          (error "Too few args supplied" vars vals))))

(define (lookup-variable-value var env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars) (env-loop (enclosing-environment env)))
            ((eq? var (car vars)) (car vals))
            (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
        (error "Unbound variable -- LOOKUP" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame) (frame-values frame))))
    (env-loop env))
```



```
(define-variable! 'true #t initial-env)
(define-variable! 'false #f initial-env)
initial-env))
(define the-global-environment (setup-environment))
```

The Read-Eval-Print Loop

```
(define input-prompt ";;; M-Eval input:")
(define output-prompt ";;; M-Eval value:")
(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output (m-eval input the-global-environment)))
      (announce-output output-prompt)
      (display output)))
    (driver-loop))
(define (prompt-for-input string)
  (newline) (newline) (display string) (newline))
(define (announce-output string)
  (newline) (display string) (newline))
```