

MASSACHUSETTS INSTITUTE OF TECHNOLOGY  
 Department of Electrical Engineering and Computer Science  
 6.001—Structure and Interpretation of Computer Programs  
 Spring Semester, 2007

**Final Exam**

**Closed Book – three sheets of notes**

Throughout this quiz, we have set aside space in which you should write your answers. Please put all of your answers in the designated spaces, as we will look only in this space when grading.

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. Also note that while there may be a lot of reading to do on a problem, there is relatively little code to write, so please take the time to read each problem carefully.

YOUR NAME:

YOUR TA:

- Matt Brown
- Austin Clements
- Michael Craig
- Stephen McCamant
- Brennan Sherry
- James Wnorowski
- Sarah Wu

PART	Value	Grade	Grader	PART	Value	Grade	Grader
1	36			5	16		
2	45			6	18		
3	20			7	15		
4	50						
				<b>TOTAL</b>	200		

No grades for 6.001 will be available until May 25th. 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, Donna Kaufman, in 38-401. 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 25th for students to look at (but not remove from) Donna Kaufman's office, Room 38-401.

**Part 1. (36/200 points): Object oriented systems**

Consider the classes below – note the methods of each class and the inheritance structure. Note that two lines of the WORK method for an entrepreneur will be changed in the questions. You should be able to answer the questions in this part just by considering the code, and using your experience in Project 4, but in case you need it, we have provided some of the object oriented system code at the end of the exam.

```
(define (lecturer self name)
  (let ((root-part (root-object self)))
    (make-handler
      'lecturer
      (make-methods
        'SAY (lambda (stuff) (display stuff) (newline))
        'NAME (lambda () name)
        'WORK (lambda () (ask self 'SAY "i love to hear myself TALK"))
        'PLAY (lambda () (ask self 'SAY (ask self 'NAME))
                      (ask self 'SAY " says: i have NO TIME to play")
                      (ask self 'WORK)))
      root-part)))

(define (professor self name)
  (let ((lecturer-part (lecturer self name)))
    (make-handler
      'professor
      (make-methods
        'WORK (lambda ()
                 (ask self 'SAY (ask self 'NAME))
                 (ask self 'SAY " says: i need to PUBLISH my research but...")
                 (ask lecturer-part 'WORK))
        'NAME (lambda () 'please-call-me-PROFESSOR)
        'PLAY (lambda () (ask self 'SAY "now that i am TENURED i can play all i want")))
      )
    lecturer-part)))

(define (entrepreneur self name)
  (let ((lecturer-part (lecturer self name))
        (professor-part (professor self name)))
    (make-handler
      'entrepreneur
      (make-methods
        'WORK (lambda ()
                 (ask self 'SAY "i've got this GREAT IDEA for a company!")
                 (ask professor-part 'WORK)) ;; HERE IS WHERE WE WILL CHANGE EXPRESSIONS
      )
    professor-part lecturer-part)
    ;; AND HERE IS WHERE WE WILL CONSIDER CHANGING ORDER
  ))

(define eric (create-instance entrepreneur 'ERIC))
```

In the following questions, we are going to consider some variations on the definition of an entrepreneur.

**Question 1:**

First, suppose we evaluate

```
(ask eric 'WORK)
```

What is printed out on the monitor? (To save writing, it's enough to write down only those words that would appear in all capital letters, SUCH AS THIS. Line breaks aren't important, but make sure the words appear in the right order.)

**Question 2:**

Now, suppose we evaluate

```
(ask eric 'PLAY)
```

What is printed out on the monitor?

**Question 3:** Now suppose we replace the last line of the WORK method for entrepreneurs with

```
(ask self 'WORK)
```

If we create a new instance and then (ask eric 'WORK), what gets printed out on the monitor?

**Question 4:** Suppose instead that we replace the last line of the WORK method with  
(ask lecturer-part 'WORK)

If we create a new instance and then (ask eric 'WORK), what gets printed out on the monitor?

**Question 5:** After making the change in the previous question, if we (ask eric 'PLAY), what gets printed out on the monitor?

**Question 6:** Suppose that we remove the entire WORK method for entrepreneurs. Now, if we create a new instance and then (ask eric 'WORK), what gets printed out on the monitor?

**Question 7:** After making the change in the previous question, if we (ask eric 'PLAY), what gets printed out on the monitor?

**Question 8:** Suppose we use the version with no `WORK` method for entrepreneurs, but we also change the order of inheritance to `lecturer-part professor-part`. Now, if we (`ask eric 'WORK`), what gets printed out on the monitor?

**Question 9:** After making the change in the previous question, if we (`ask eric 'PLAY`), what gets printed out on the monitor?

**Part 2 (45/200 points): A Model of the Web**

The focus of this part is to model the World Wide Web using Scheme data structures.

At the most basic level, a web page contains *words*, which we will represent by symbols.

Every page also has a name, called a *URL*, which we will also represent by a symbol. URL symbols will start with `http://` by convention.

A page also contains *links*. On a real web page, a link is an underlined word that points to another web page. In our Scheme model, a link is represented by a tagged data abstraction containing a word (which is what's underlined) and a URL (the page the link goes to when you click on it).

A page is a tagged data abstraction containing the page's URL and a list of the words and links on the page. For example, here is Ben Bitdiddle's home page:

```
(page http://mit.edu/benbitdiddle
  (I am taking (link 6.001 http://sicp.mit.edu/) this semester
    and I like (link Salsa http://salsa.mit.edu/) dancing.))
```

And here is the 6.001 homepage that one of Ben's links points to:

```
(page http://sicp.mit.edu/
  (Welcome to 6.001
    The first project is (link here http://sicp.mit.edu/project1)))
```

Procedures for the page and link abstractions are shown below. Note that the `data` argument for `make-page` has type `List<symbol or link>`.

```
(define (make-page url data) (list 'page url data))
(define (page? object) (and (pair? object) (eq? 'page (car object))))
(define (page-url page) (cadr page))
(define (page-data page) (caddr page))

(define (make-link word url) (list 'link word url))
(define (link? object) (and (pair? object) (eq? 'link (car object))))
(define (link-word link) (cadr link))
(define (link-url link) (caddr link))
```

**Question 10:**

We want a simple way to add hyperlinks to a page, by replacing all occurrences of a particular word, such as 6.001, with a hyperlink, such as to the 6.001 page.

Write a higher-order procedure `replace!` that takes a predicate procedure `pred` and applies it to every element of a list. For every element for which `pred` returns true, the element should be passed to a transformer procedure `proc`, and the element should be replaced in the list by the value returned by `proc`. The list should be mutated rather than copied.

For example, if `x` is bound to the list `(1 2 3 4)`, then `(replace! even? square x)` should square every even element of the list, so that afterwards `x` is bound to `(1 4 3 16)`.

Fill in the definition of `replace!` below. Do **not** use existing higher-order procedures like `map` or `filter` in your solution.

```
(define (replace! pred proc  
  lst)
```

**Question 11:**

Now write a procedure `add-links!` that takes a page, a word, and a url and replaces every occurrence of the word in the page with a hyperlink to that url. For example:

```
ben-dance-page
=> (page http://mit.edu/ben/dance
      (Salsa Contra Tango but Salsa is my favorite))
```

```
(add-links! ben-dance-page 'Salsa 'http://salsa.mit.edu)
```

```
ben-dance-page
=> (page http://mit.edu/ben/dance
      ((link Salsa http://salsa.mit.edu) Contra Tango but
       (link Salsa http://salsa.mit.edu) is my favorite))
```

Fill in the definition of `add-links!` below:

```
(define (add-links! page word url)
  (replace! ANSWER-A))
```

Write the expressions that should replace ANSWER-A in the box below.

**Question 12:**

Search engine crawlers move through the web by following links from one page to another, so they need to find all the links in a page. Write a procedure `outgoing-urls` that takes a page and returns the list of URLs found in the links on the page. For example,

```
ben-dance-page
=> (page http://mit.edu/ben/dance
    ((link Salsa http://salsa.mit.edu) Contra Tango but
     (link Salsa http://salsa.mit.edu) is my favorite))

(outgoing-urls ben-dance-page)
=> (http://salsa.mit.edu http://salsa.mit.edu)
```

Fill in the definition of `outgoing-urls` below, which uses `map` and `filter`.

```
(define (outgoing-urls page)
  (map ANSWER-B
    (filter ANSWER-C ANSWER-D)))
```

ANSWER-B:

ANSWER-C:

ANSWER-D:

**Question 13:**

Now we want to create a simple web browser, allowing the user to click around our simple model of the web. To do that, we'll need a procedure for clicking on a link.

We want a procedure `click` that takes a page and a number  $n$ , finds the  $n$ th link on the page (where the first link is  $n = 0$ ) and returns its URL. If  $n$  is too large for the number of links on the page, then `click` should return the symbol `'not-found`. For example:

```
ben-homepage
=> (page http://mit.edu/benbitdiddle
    (I am taking (link 6.001 http://sicp.mit.edu/) this semester
    and I like (link Salsa http://salsa.mit.edu/) dancing.))

(click ben-homepage 0)
=> http://sicp.mit.edu/

(click ben-homepage 1)
=> http://salsa.mit.edu/

(click ben-homepage 2)
=> not-found
```

Fill in the definition for `click` below.

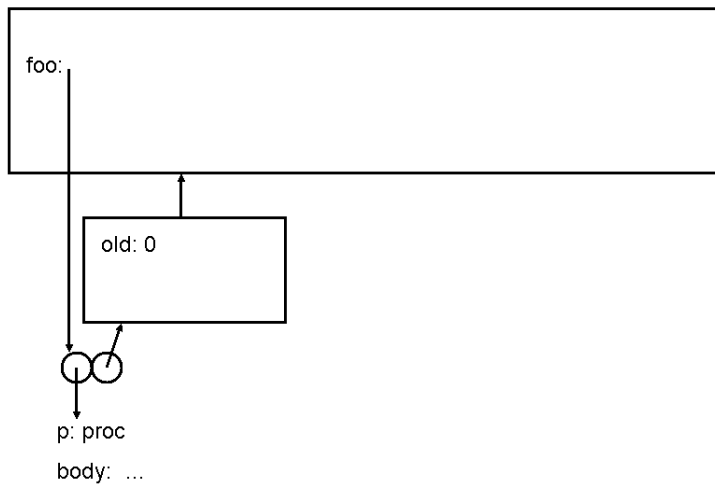
```
(define (click page n)
  ANSWER-E)
```

ANSWER-E:

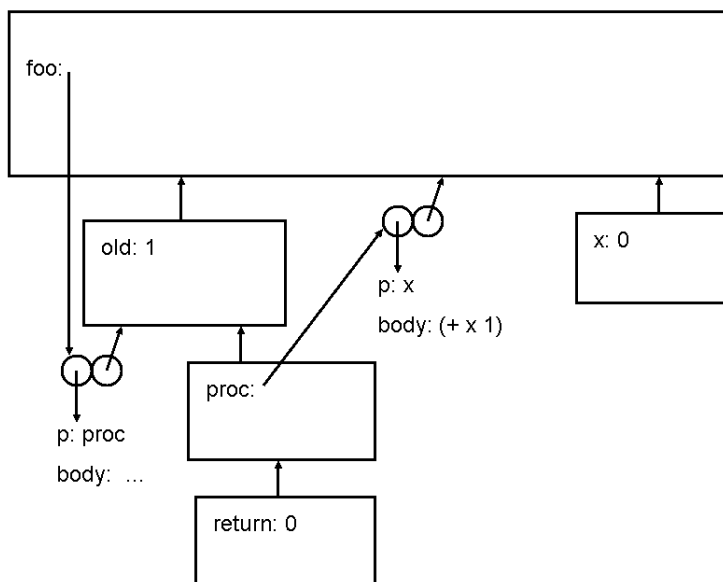
**Part 3 (20/200 points): Environment model**

Consider the following environment diagrams. The first shows the state of the environment, after evaluation of a completion of the following expression:

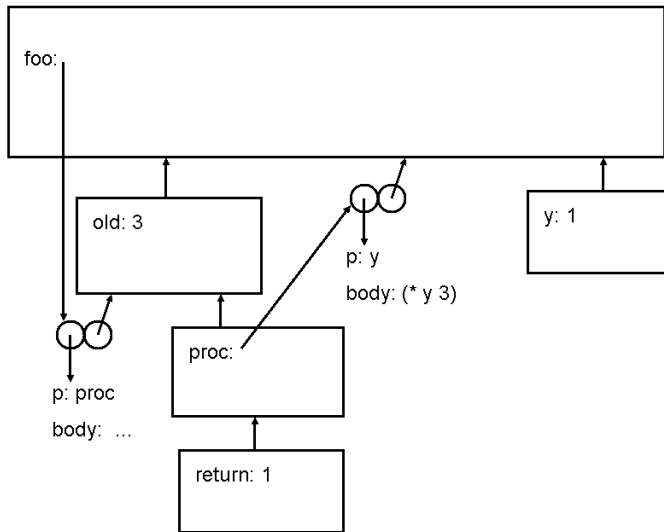
```
(define foo TO-BE-COMPLETED)
```



The second shows the state of the environment after an application of `foo`. The application returns 0 as its value.



The third shows the state of the environment after a second, subsequent application of `foo`. The application returns 1. (Note that we have removed parts of the diagram from the first application that are no longer relevant.)



**Question 14:**

Complete the definition of `foo` so that it is consistent with the environment diagrams:

**Question 15:**

What expression involving an application of `foo` would lead to the second diagram:

**Question 16:**

What expression involving an application of `foo` would lead to the third diagram:

**Part 4 (50/200 points): Evaluators**

Suppose that we want to add a new special form, called `loop` to our meta-circular evaluator. An example of this special form is shown below:

```
(loop ((i 1 (+ 1 i))
      (ans 1 (* ans i)))
      ((= i n) ans)
      (display i)
      (display ans))
```

The syntax of a `loop` is as follows:

- The first subexpression is a sequence of **clauses**. Each clause has three parts, a **variable**, an expression whose value determines the variable's **initial value**, and an expression whose value is used to **update** the variable on each iteration through the loop. In the example above, there are two clauses, one with variable `i`, the second with variable `ans`.
- The second subexpression has two parts: an **end test** and a **return** expression.
- The remainder of the `loop` expression is a sequence of expressions, called the **body**.

The semantics of a `loop` is the following:

1. First, the initial expressions are evaluated as a group, and then each variable is bound to the value of its initial expression (similar to how `let` works).
2. Next, the end test is evaluated. If it is **true**, then the **return** expression is evaluated and its value is returned as the value of the whole loop.
3. Otherwise, each expression in the body is evaluated.
4. Finally, the **update** expressions are evaluated as a group, and then each variable is bound to the value of its update expression (again, similar to `let`). The process is then repeated from step 2.

**Question 17:**

First, we need to create some syntax handlers. For each part, provide the definition:

**loop-clauses**: takes as argument a `loop` expression and returns the list of clauses; in the above example, it would return `((i 1 (+ 1 i)) (ans 1 (* ans i)))`

```
(define loop-clauses )
```

**loop-end**: takes as argument a `loop` expression and returns the list of end test and return expression; in the above example, it would return `((= i n) ans)`

```
(define loop-end )
```

**loop-body**: takes as argument a **loop** expression and returns the list of body expressions; in the above example, it would return `((display i) (display ans))`

```
(define loop-body )
```

**clause-var**: takes as argument a **clause** expression and returns the variable part; in the above example, if applied to the first expression of the output of **loop-clauses** it would return `i`

```
(define clause-var )
```

**clause-init**: takes as argument a **clause** expression and returns the initial value part; in the above example, if applied to the first expression of the output of **loop-clauses** it would return `1`

```
(define clause-init )
```

**clause-update**: takes as argument a **clause** expression and returns the update part; in the above example, if applied to the first expression of the output of **loop-clauses** it would return `(+ 1 i)`

```
(define clause-update )
```

**end-test**: takes as argument the output of **loop-end**, and returns the end test expression; in the above example, it would return `(= i n)`

```
(define end-test )
```

**end-return**: takes as argument the output of **loop-end**, and returns the end return expression; in the above example, it would return `ans`

```
(define end-return )
```

When we add a new special form to our evaluation procedure, `m-eval`, we check for the special form, then call a special evaluation procedure. The procedure to check for a `loop` expression is:

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

If applying this procedure evaluates to true, then the procedure `eval-loop` is called, as in this excerpt from `m-eval`:

```
(define (m-eval exp env)
  (cond ((self-evaluating? exp) exp)
        ...
        ((loop? exp) (eval-loop exp env))
        ...
        ((application? exp)
         (m-apply (m-eval (operator exp) env)
                   (list-of-values (operands exp) env)))
        (else (error "Unknown expression type -- EVAL" exp))))
```

For your convenience, a copy of the Meta-Circular Evaluator is attached at the end of the exam.

### Question 18:

Here is a partially completed version of this procedure:

```
(define (eval-loop exp env)
  (let ((clauses (loop-clauses exp))
        (end (loop-end exp))
        (body (loop-body exp)))
    (let ((vars (map clause-var clauses))
          (inits (map clause-init clauses))
          (updates (map clause-update clauses)))
      (let ((new-env (extend-environment
                     ANSWER-A
                     ANSWER-B
                     ANSWER-C)))
        (loopit vars updates end body new-env))))))
```

`new-env` should contain bindings of the variables to the initial values for each clause. Provide expressions for:

ANSWER-A

ANSWER-B

ANSWER-C

Finally, you need to implement `loopit`

```
(define (loopit vars updates end body env)
  ANSWER-D)
```

ANSWER-D



**Question 19:**

Now suppose that we want to add a new special form `assert` to our meta-circular evaluator. Our new form `assert` evaluates a predicate expression. If the predicate evaluates to true, then the value of the `assert` expression is simply true. If the predicate evaluates to false, then `assert` should signal an error, displaying a message that includes the predicate expression that failed (*not* its value). For example:

```
(define x 3)
(assert (> x 0)) ; => evaluates to #t
(assert (< x 0)) ; => signals an error that displays
Assertion failed: (< x 0)
```

To signal an error, use the `error` procedure. For example, `(error "Your lucky number is " 5)` would signal an error and display the message `Your lucky number is 5`.

Recall that one way to add new forms to our Meta-Circular evaluator is to create syntactic transformations, in which a special form is converted, using manipulation of the syntactic representation (e.g., 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.

Assume that we have syntax procedures for detecting and pulling apart an `assert` expression:

- `(assert? exp)` tests whether `exp` is an `assert` expression
- `(assert-predicate exp)` returns the predicate expression of the `assert` expression

Add a syntactic transformation for handling `assert` expressions, by converting an `assert` expression into an `if` expression. Hence, the new clause in `m-eval` for handling `assert` would be

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

Provide code that completes the definition of `assert->if`.

```
(define (assert->if exp) ANSWER-E)
```

ANSWER-E

**Part 5 (16/200 points): Streams**

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

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

VERSION 1:

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

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

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

VERSION 2:

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

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

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

```
(define (print-stream-n st n)
  (if (< n 1)
      'done
      (begin
        (display (stream-car st)) (display " ")
        (print-stream-n (stream-cdr st) (- n 1))))))
```

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

Possible outcomes:

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

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

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

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

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

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

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

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

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

```
(print-stream-n st 5)  
1 7 8 9 10 done
```

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

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

```
ERROR                          ;;; OUTCOME G
```

```
None of the above              ;;; OUTCOME H
```

List the outcome (chosen from A, B, C, D, E, F, G, or H) that would be produced in each of the following cases.

**Question 20:** Version 1; no memoization:

**Question 21:** Version 1; with memoization:

**Question 22:** Version 2; no memoization:

**Question 23:** Version 2; with memoization:

**Part 6 (18/200 points): Orders of Growth**

For each of the following procedures, indicate its order of growth:

- in time, as measured by the number of primitive operations to be performed (call this  $T(n)$ );
- in space on the stack, as measured by the maximum number of deferred operations that are stored on the stack during evaluation (call this  $S(n)$ );
- in space outside the stack, as measured by the total number of `cons` cells created during the evaluation (call this  $H(n)$ ).

For each function,  $n$  is the size of the input, which is either the length of the list parameter or the size of the integer parameter. Assume that any procedures passed as arguments are primitive procedures that do not create any new `cons` cells.

For each of the following questions, choose the description from these options that best describes the order of growth of the process.

- A: constant
- B: linear
- C: exponential
- D: quadratic
- E: logarithmic
- F: something else

**Question 24:**

```
(define (map proc lst)
  (if (null? lst)
      '()
      (cons (proc (car lst))
            (map proc (cdr lst)))))
```

$T(n)$ :   $S(n)$ :   $H(n)$ :

**Question 25:**

```
(define (map2 proc lst)
  (define (helper rest answer)
    (if (null? rest)
        answer
        (helper (cdr rest)
                 (append answer
                          (list (proc (car rest)))))))
  (helper lst '()))
```

where `append` is defined by

```
(define (append lst1 lst2)
  (if (null? lst1)
      lst2
      (cons (car lst1)
            (append (cdr lst1) lst2))))
```

$T(n)$ :   $S(n)$ :   $H(n)$ :

**Question 26:**

```
(define (f n)
  (cond ((= n 0) '())
        ((= n 1) (list 1))
        (else (list (f (- n 1)) (f (- n 2))))))
```

$T(n)$ :   $S(n)$ :   $H(n)$ :

**Part 7 (15/200 points): General questions**

Fill in the blank in each of the following statements.

**Question 27:**

A last-in, first-out (LIFO) data structure is called a

**Question 28:**

Saying a language has  means that a procedure body is evaluated in an environment that extends the environment where the procedure was applied.

**Question 29:**

Saying a language has  means that operand expressions to a procedure are evaluated only when the operand's value is actually needed.

**Question 30:**

Languages with  can implement iterative algorithms using recursive procedures.

**Question 31:**

Functional programming means programming without

**Question 32:**

A  is defined as a procedure that takes a procedure as argument, returns a procedure as value, or both.

END OF EXAM

## 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))

  (env-loop env))

(define (set-variable-value! var val env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars) (env-loop (enclosing-environment env)))
            ((eq? var (car vars))
             (set-car! vals val)) ; Same as lookup except for this
```

```

        (else (scan (cdr vars) (cdr vals))))))
    (if (eq? env the-empty-environment)
        (error "Unbound variable -- SET!" var)
        (let ((frame (first-frame env)))
            (scan (frame-variables frame) (frame-values frame))))))
(env-loop env))

(define (define-variable! var val env)
  (let ((frame (first-frame env)))
    (define (scan vars vals)
      (cond ((null? vars) (add-binding-to-frame! var val frame))
            ((eq? var (car vars)) (set-car! vals val))
            (else (scan (cdr vars) (cdr vals)))))
    (scan (frame-variables frame)
          (frame-values frame))))

```

## Primitive Procedures and the Initial Environment

```

(define (primitive-procedure? proc) (tagged-list? proc 'primitive))
(define (primitive-implementation proc) (cadr proc))
(define primitive-procedures
  (list (list 'car car)
        (list 'cdr cdr)
        (list 'cons cons)
        (list 'null? null?)
        (list '+ +)
        (list '> >)
        (list '= =)
        (list '* *)
        (list 'display display)
        (list 'not not)
        ; ... more primitives
  ))

(define (primitive-procedure-names) (map car primitive-procedures))
(define (primitive-procedure-objects)
  (map (lambda (proc) (list 'primitive (cadr proc))) primitive-procedures))
(define (apply-primitive-procedure proc args)
  (apply (primitive-implementation proc) args))

(define (setup-environment)
  (let ((initial-env (extend-environment (primitive-procedure-names)
                                       (primitive-procedure-objects)
                                       the-empty-environment)))
    (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))
```

## Object-Oriented System

Selected code from object oriented system in Project 4.

```
;;-----
;; Instance

; instance is a tagged data structure which holds the ‘‘self’’ of a
; normal object instance. It passes all messages along to the handler
; procedure that it contains.

(define (make-instance)
  (list 'instance #f))

(define (instance? x)
  (and (pair? x) (eq? (car x) 'instance)))

(define (instance-handler instance) (cadr instance))

(define (set-instance-handler! instance handler)
  (set-car! (cdr instance) handler))

(define (create-instance maker . args)
  (let* ((instance (make-instance))
        (handler (apply maker instance args)))
    (set-instance-handler! instance handler)
    (if (method? (get-method 'INSTALL instance))
        (ask instance 'INSTALL))
    instance))

;;-----
;; Handler

; handler is a procedure which responds to messages with methods
; it automatically implements the TYPE and METHODS methods.

(define (make-handler typename methods . super-parts)
  (cond ((not (symbol? typename)) ;check for possible programmer errors
        (error "bad typename" typename))
        ((not (method-list? methods))
         (error "bad method list" methods))
        ((and super-parts (not (filter handler? super-parts)))
         (error "bad part list" super-parts))
        (else
         (let ((handler
                (lambda (message)
                  (case message
                    ((TYPE)
                     (lambda () (type-extend typename super-parts)))
                    ((METHODS)
                     (lambda ()
                      (append (method-names methods)
                              (append-map (lambda (x) (ask x
                                                    'METHODS))
                                           super-parts)))))))
           handler))))))
```

```

                (else
                  (let ((entry (method-lookup message methods)))
                    (if entry
                      (cadr entry)
                      (find-method-from-handler-list message
                                                       super-parts)))))))))
    handler
  ))))

(define (handler? x)
  (procedure? x))

(define (->handler x)
  (cond ((instance? x)
        (instance-handler x))
        ((handler? x)
         x)
        (else
         (error "I don't know how to make a handler from" x))))

; builds a list of method (name,proc) pairs suitable as input to
; make-handler
; note that this puts a label on the methods, as a tagged list

(define (make-methods . args)
  (define (helper lst result)
    (cond ((null? lst) result)

          ; error catching
          ((null? (cdr lst))
           (error "unmatched method (name,proc) pair"))
          ((not (symbol? (car lst)))
           (if (procedure? (car lst))
               (pp (car lst))
               (error "invalid method name" (car lst))))
          ((not (procedure? (cadr lst)))
           (error "invalid method procedure"
                  (cadr lst))))

          (else
           (helper (cddr lst) (cons (list
                                     (car lst) (cadr lst)) result))))))
  (cons 'methods (reverse (helper args '()))))

(define (method-list? methods)
  (and (pair? methods) (eq? (car methods) 'methods)))

(define (empty-method-list? methods)
  (null? (cdr methods)))

(define (method-lookup message methods)
  (assq message (cdr methods)))

(define (method-names methods)

```

```

(map car (cdr methods)))

;;-----
;; Root Object

; Root object. It contains the IS-A method.
; All classes should inherit (directly or indirectly) from root.
;
(define (root-object self)
  (make-handler
   'ROOT
   (make-methods
    'IS-A
    (lambda (type)
      (memq type (ask self 'TYPE))))))

;;-----
;; Object Interface

; ask
;
; We "ask" an object to invoke a named method on some arguments.
;
(define (ask object message . args)
  ;; See your Scheme manual to explain '. args' usage
  ;; which enables an arbitrary number of args to ask.
  (let ((method (get-method message object)))
    (cond ((method? method)
           (apply method args))
          (else
           (error "No method for" message 'in
                  (safe-ask 'UNNAMED-OBJECT
                            object 'NAME))))))

; Safe (doesn't generate errors) method of invoking methods
; on objects. If the object doesn't have the method,
; simply returns the default-value. safe-ask should only
; be used in extraordinary circumstances (like error handling).
;
(define (safe-ask default-value obj msg . args)
  (let ((method (get-method msg obj)))
    (if (method? method)
        (apply ask obj msg args)
        default-value)))

;;-----
;; Method Interface
;;
;; Objects have methods to handle messages.

; Gets the indicated method from the object or objects.
; This procedure can take one or more objects as
; arguments, and will return the first method it finds
; based on the order of the objects.

```

```
;
(define (get-method message . objects)
  (find-method-from-handler-list message (map ->handler objects)))

(define (find-method-from-handler-list message objects)
  (if (null? objects)
      (no-method)
      (let ((method ((car objects) message)))
        (if (not (eq? method (no-method)))
            method
            (find-method-from-handler-list message (cdr
                                                    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)))

; used in make-handler to build the TYPE method for each handler
;
(define (type-extend type parents)
  (cons type
        (remove-duplicates
         (append-map (lambda (parent) (ask parent 'TYPE))
                     parents))))

(define (append-map proc lst)
  (apply append (map proc lst)))
```