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

Note that any procedures or code fragments that you write will be judged not only on correct function, but also on clarity and good programming practice.

The answer sheet asks for your section number and tutor. For your reference, here is the table of section numbers.
Part 1: (18 points)

Consider the following procedure:

```
(define (last-call proc)
  (let ((old '*not-used*))
    (lambda (arg)
      (let ((temporary old))
        (set! old (proc arg))
        temporary)))
```

For example, we might have:

```
(define weird (last-call (lambda (x) (* x x))))
```

```
(weird 5)
;Value: *not-used*
```

```
(weird 7)
;Value: 25
```

Suppose we actually evaluate the above sequence of expressions in the order shown:

We want you to draw the environment diagram generated by these expressions, using diagram fragments that we provide. Attached to the exam is a tear-off sheet, with some fragments from an environment diagram. In this diagram, we have marked procedure objects as P1, P2, etc., environments as E1, E2, etc.

You should EITHER complete this diagram directly on these fragments, OR you should do your own environment diagram on a separate sheet, then copy the labels for the fragments onto your diagram (we actually recommend the latter). **In either case, answer the questions about the environment based on the labels used on OUR diagram!!**

First, for each procedure object, P1 through P4, identify, if possible, the environment pointer of the procedure (i.e. one of GE, E1, ..., E8). If the appropriate environment is not shown, write “not shown”.

**Question 1.** To what does the environment pointer of P1 point?

**Question 2.** To what does the environment pointer of P2 point?

**Question 3.** To what does the environment pointer of P3 point?

**Question 4.** To what does the environment pointer of P4 point?

Second, for each environment frame, indicate which environment is the enclosing environment for that frame. If the appropriate environment is not shown, write “not shown”. If there is no environment, write “none”.

**Question 5.** What is the enclosing environment for E1?
Question 6. What is the enclosing environment for E2?

Question 7. What is the enclosing environment for E3?

Question 8. What is the enclosing environment for E4?

Question 9. What is the enclosing environment for E5?

Question 10. What is the enclosing environment for E6?

Question 11. What is the enclosing environment for E7?

For each of the following questions, choose from among these possibilities:

- one of the procedure objects, P1, ..., P5,
- a number (say what number)
- a symbol (say what specific symbol)
- a list of numbers or symbols or procedure objects (which you must draw as box-and-pointer notation),
- an environment, E1, ..., E7, GE,
- nothing

Do this in terms of the values associated with these variables after all the expressions have been evaluated.

Question 12. To what is the variable last-call in the global environment bound?

Question 13. To what is the variable proc in E7 bound?

Question 14. To what is the variable old in E6 bound?

Question 15. What variable is bound to the procedure object P4? Give both the name and the environment, if applicable.

Question 16. What variable is bound to the procedure object P5? Give both the name and the environment, if applicable.
Part 2 (18 points):

At the end of the quiz you will find the code for our object oriented system.

The following object oriented class hierarchy covers different ways objects can hold state. Look over the code carefully before answering the questions.

```
(define (make-A state)
  (lambda (msg)
    (case msg
      ((GETSTATE) (lambda (self) state))
      ((ALTER) (lambda (self newstate)
                (set! state newstate)
                (ask self 'GETSTATE)))
      ((REPORT) (lambda (self)
                (list (ask self 'GETSTATE)))
      (else (no-method))))))

(define (make-B state)
  (let ((my-A (make-A state)))
    (lambda (msg)
      (case msg
        ((ALTER) (lambda (self newstate)
                (set! state newstate)
                (ask self 'GETSTATE)))
        ((REPORT) (lambda (self)
                (cons state (ask my-A 'REPORT)))
      (else (get-method msg my-A))))))

(define (make-C state)
  (let ((my-A (make-A state))
        (my-B (make-B state)))
    (lambda (msg)
      (case msg
        ((GETSTATE) (lambda (self) (ask my-A 'GETSTATE)))
        ((ALTER) (lambda (self newstate)
                (ask my-A 'ALTER newstate)))
        ((REPORT) (lambda (self)
                (delegate my-B self 'REPORT)))
      (else (get-method msg my-A))))))
```
Assume we make the above definitions, and then evaluate

\[
\begin{align*}
&\text{(define alex (make-A 'alpha))} \\
&\text{(define bruce (make-B 'beta))} \\
&\text{(define chris (make-C 'gamma))}
\end{align*}
\]

What gets returned as value for each of the following expressions? (Assume that they are evaluated in this order.) You may find it helpful to draw an instance diagram or a class diagram to keep track of the state and structure of the objects created in this example.

Question 17.

(ask alex 'REPORT)

Question 18.

(ask alex 'ALTER 'alef)

Question 19.

(ask alex 'REPORT)

Question 20.

(ask bruce 'REPORT)

Question 21.

(ask bruce 'ALTER 'betel)

Question 22.

(ask bruce 'REPORT)

Question 23.

(ask chris 'REPORT)

Question 24.

(ask chris 'ALTER 'gum)

Question 25.

(ask chris 'REPORT)
Part 3 (20 points)

We are going to add a new special form to our meta-circular evaluator (a copy of which is attached at the end of the quiz). The form we are going to add is an unless expression, such as

```
(define test '(1 2 3))
(unless (null? test)
  (display (car test))
  (set! test (cdr test)))
```

The idea is that an unless expression consists of two parts. The first sub-expression is a test clause, the remaining sub-expressions are the body. The evaluation of an unless is to first evaluate the test. If it is true, then evaluation of the entire unless is terminated, with an unspecified return value. Otherwise, the body of the unless is evaluated and the process repeats.

To the meta-circular evaluator, we add the following clause

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

where

```
(define (unless? exp) (tagged-list? exp 'unless))
(define (unless-test exp) (cadr exp))
(define (unless-body exp) (cddr exp))
```

**Question 26.** Write the procedure `eval-unless` by completing the following definition. Be sure to use the appropriate data abstractions.

```
(define (eval-unless exp env)
  COMPLETE)
```

**Question 27.** Suppose instead that we add the following clause to the evaluator:

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

Complete the syntactic transformation

```
(define (unless->if exp)
  COMPLETE)
```

so that this will correctly evaluate unless expressions.
Part 4: (22 points)

We want to construct a data abstraction called a queue. A queue consists of an ordered sequence of items, which we will represent using a list. Users of queues see a data abstraction, with a queue consisting of two components, a head and a tail. One can read the element at the head of the queue, but cannot directly access other elements in the queue. Attempting to read an empty queue is an error.

One can only add new elements to a queue at the tail of the queue, and one can only remove elements from the head of the queue.

For example, here is some code to create queues, and some interactions with a queue.

```scheme
(define (make-queue elt)
  (let ((temp (list elt)))
    (cons temp temp)))
(define (head q) (car q))
(define change-head! set-car!)
(define (tail q) (cdr q))
(define change-tail! set-cdr!)
(define (read q) (car (head q)))
(define (add-queue elt q)
  (let ((new-part (list elt)))
    (cond ((null? (tail q))
       INSERT-1)
       (else INSERT-2
           INSERT-3)))
q)
(define (delete-queue q)
  (cond ((eq? (head q) (tail q))
     (change-head! q #f)
     (change-tail! q #f))
     (else INSERT-4))
q)
;; HERE IS AN EXAMPLE QUEUE
(define my-queue (make-queue 1))
;Value: ((1) 1)
(read my-queue)
;Value: 1

Question 28: Draw a box and pointer diagram for the value of my-queue.
Now, consider the following actions on a queue:

\[
\begin{align*}
&\text{(add-queue 2 my-queue)} \\
&;\text{Value: ((1 2) 2)} \\
&(\text{read my-q)} \\
&;\text{Value: 1} \\
&;\text{; NOTE: the same value is still at the head} \\
&\text{(add-queue 3 my-queue)} \\
&;\text{Value: ((1 2 3) 3)} \\
&\text{(delete-queue my-queue)} \\
&;\text{Value: ((2 3) 3)} \\
&(\text{read my-q)} \\
&;\text{Value: 2} \\
&;\text{; NOTE: the value under the head has now changed} \\
&\text{(delete-queue my-queue)} \\
&;\text{Value: ((3) 3)} \\
&(\text{delete-queue my-queue)} \\
&;\text{Value: (#f)} \\
&;\text{; NOTE: we are now left with an empty queue}
\end{align*}
\]

**Question 29:** Provide code for INSERT-1, which should create a queue with one element.

**Question 30:** Provide code for INSERT-2, so that the internal representation of the elements of the queue is modified correctly.

**Question 31:** Provide code for INSERT-3, so that the external representation of the queue is modified correctly.

**Question 32:** Provide code for INSERT-4.

Now suppose we decide to change the underlying representation of a queue, to use a list rather than a pair to represent the head and tail of the queue.

\[
\begin{align*}
&\text{(define (make-queue elt)} \\
&\quad \text{(let ((temp (list elt)))} \\
&\quad\quad \text{(list temp temp)}))} \\
&;\text{NOTE THE CHANGE HERE}
\end{align*}
\]

**Question 33:** What is the minimal set of definitions that must change for the system to still operate correctly? Provide the correct new definitions.
Part 5 (22 points): Attached at the end of the quiz is a copy of the meta-circular evaluator. As it presently stands, this version of the evaluator cannot deal with procedures that use the “dot” notation for their arguments, such as:

```
(define foo
  (lambda (x . y)
    (if (> x 0) (length y))))
```

```
(foo 1 2 3 4)
;Value: 3
```

```
(foo 1)
;Value: 0
```

```
(foo 2 5)
;Value: 1
```

Remember that the idea behind the “dot” notation was to allow a procedure to be called with variable numbers of arguments. In this example, `foo` must be called with at least one argument, and the formal parameter `x` would be bound to that value.

However, `foo` could also be called with 2 or more arguments, and the formal parameter `y` would then be bound to a list of the values of all the subsequent arguments. Thus in

```
(foo 1 2 3 4)
```

the formal parameter `x` would be bound to the value 1 and the formal parameter `y` would be bound to the list `(2 3 4)`.

When we evaluate a `lambda` expression, it will create a representation for a procedure (see `make-procedure` in the evaluator code). In particular, when the reader converts the input expression such as `(lambda (x y) ...)` into an internal representation for the evaluator, it provides list structure like this:

```
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
\|/    \|/
```

whereas if the reader is given input such as `(lambda (x . y) ...)`, this is converted into list structure like this:

```
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
\|/    \|/
```

\```
X       Y
```
Thus when we evaluate a lambda expression whose parameter list is of the ordinary sort (e.g., (x y) or (x y z a)), the internal representation of this parameter list is an ordinary list (i.e., a sequence of cons cells whose cdr-pointers end in nil).

However, when the parameter list includes a “dot” (such as (x . y) or (x y z . a)), the representation for this parameter list will not be an ordinary list. It will differ in one small but important detail: the cdr of the last cons cell in the sequence will point to the last parameter (rather than to nil).

To change our evaluator to handle “dot” notation, we just need to modify extend-environment:

```
(define (extend-environment vars vals base-env)
  (let ((processed-lists (find-dot vars vals)))
    (cons (make-frame (car processed-lists)
                      (cadr processed-lists))
          base-env)))
```

The find-dot procedure walks down the sequence of variable names and associated values, checking to see whether vars is an ordinary list (i.e., no “dot”) or whether it is a sequence of cons cells that terminates by pointing at a variable rather than at nil (i.e., it had a dot). When we find that the cdr of the last cons cell points to a variable name rather than to nil, we want to

(a) add that variable to the end of a list of variables, and
(b) add the remaining values as a list, which should be added to the end of the list of all values.

(Notice that we still add the variable name to a list of variables, but handle the remaining values differently.)

Finally, find-dot will return a list of two lists – one for the variables, and one for the values.

Here is a template for this:

```
(define (find-dot vars vals)
  (define (help vars-seen vars-todo vals-seen vals-todo)
    (cond ((null? vars-to-do) ; no more variables to examine
           (if (null? vals-to-do) ; if no more values either
                vals-seen
                (if (null? vals-to-do) ; if no more values either
                    vals-seen
                    (help vars-seen vars-seen vals-seen vals-todo))))
      ((null? vars-to-do) ; no more variables to examine
       vals-seen
       (if (null? vals-to-do) ; if no more values either
           vals-seen
           (help vars-seen vars-seen vals-seen vals-todo))))
```
Question 34:
What code should be used for `INSERT-5`?

Question 35:
What code should be used for `INSERT-6`?

Question 36:
What code should be used for `INSERT-7`?
BACKGROUND INFORMATION—THIS PAGE CONTAINS NO QUESTIONS

; Our familiar OOP system

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

(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 (safe-ask 'unnamed-object in-object 'NAME)))
          (else (error "Can't delegate" message
                       "from" (safe-ask 'unnamed-object for-object 'NAME)
                       "to" (safe-ask 'unnamed-object in-object 'NAME))))))

(define (safe-ask default-value obj msg . args)
  (let ((method (get-method msg obj)))
    (if (method? method)
      (apply-method obj obj msg args)
      default-value)))

(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)))
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-quotuation 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) (cddr exp)))) ; formal params, body

(define (lambda? exp) (tagged-list? exp 'lambda))
(define (lambda-parameters lambda-exp) (cadr lambda-exp))
(define (lambda-body lambda-exp) (cddr 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? (cdddr exp))) (cadddr exp) 'false))
(define (make-if pred conseq alt) (list 'if pred conseq 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 (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))