# CSE 341 — Assignment 2
# October 27, 1998 — Solution

1. Add primitives for arithmetic (+, -, *, and /). Demonstrate that your augmented interpreter works correctly for these new primitives. Note: it's OK to implement a version of + that takes an indefinite number of arguments, or exactly two arguments. Actually, though, you would have to go out of your way to restrict it to exactly two arguments. (Why?)

   **Solution:** The solution to this problem is to add the following to the list `primitive-procedures`

   ```
   (define primitive-procedures
     (list (list 'car car)
           ...
   ;;; BEGIN: added for problem 1
           (list '+ +)
           (list '* *)
           (list '- -)
           (list '/ /)
   ;;; END: added for problem 1
           ...
           ))
   ```

   Now, when the meta interpreter encounters an application of some values, say `'(1 2 3)`, to the `+` operator, it will invoke the regular scheme interpreter's `+` operator.

   There are several ways of doing this that limits the number of arguments to `+` For example you could have replaced the `(list '+ +)` above with `(list '+ (lambda (x y) (+ x y)))`. There are other alternatives that are even more difficult. These solution are all more difficult than that given above.

   **Output:**

   ```
   ;;; M-Eval input:
   (- 1)

   ;;; M-Eval value:
   -1

   ;;; M-Eval input:
   (+ 4 5)

   ;;; M-Eval value:
   9

   ;;; M-Eval input:
   (* 10 11 12)
   ```

```
;;; M-Eval value:
1320

;;; M-Eval input:
(/ 1 10)

;;; M-Eval value:
0.1
```

2. Add a trace facility to the interpreter. If you have defined a function factorial (in the interpreter) and then evaluate `(my-trace factorial)`[1] your interpreter should print out trace information each time factorial is called and each time it exits. On function entry, print out the name of the function and the arguments; on exit print out the name again and the return value. `(my-untrace factorial)` should turn off tracing.

   `factorial` **Solution:**

```
;; factorial
(define (factorial n)
   (if (= 0 n)
       1
       (* n (factorial (- n 1)))
   )
)
```

   This relies on adding a new primitive `=` which we added to the `primitive-procedures` list in the same manner as Problem 1.

   **Detecting a traced procedure:** To detect a traced procedure we modified the procedure data. Recall that a procedure, for example `(lambda (x) (+ 5 x))`, is represented as follows:

```
'(procedure (x) ((+ 5 x)) <procedure-env>)
```

   Well, we will make a traced procedure as follows:

```
'(traced-procedure (x) ((+ 5 x)) <procedure-env>)
```

   Now, we are all set to write code to detect if a procedure is a traced procedure[2]:

```
(define (traced-procedure? p)
  (tagged-list? p 'traced-procedure)
)
```

   We also modified `compound-procedure?` as follows:

[1] It does not matter whether you called your trace function `my-trace` or `trace`. Since it is only defined in the meta-interpreter's environment it does not interfere with the normal scheme trace function.

[2] This code was modified a little bit for Problem 3

```
(define (compound-procedure? p)
  (or (tagged-list? p 'procedure) (traced-procedure? p))
)
```

because a traced procedure is still a compound procedure.

Note that an equally good solution would be to keep a global list of traced procedures and when `my-trace` is called on some lambda, that procedure can be added to the global list of traced procedures. When `my-untrace` is called on that lambda, the global list can be searched and that procedure can be removed.

**Getting the procedure's name:** This is also rather difficult. Since procedures (or lambdas) are first class objects in scheme one lambda can be bound to any number of variable names. A little experimentation with the `trace` function in the regular scheme interpreter yields that when a procedure is traced, only the first name it was bound to gets displayed as the function name. For instance

```
(define (foo x) (+ x 5))
(define bar foo)
(define foo 55)
(define trace bar)
(bar 10)
```

Here `bar` will be traced and it will think that its function name is `foo`. With this model in mind the solution that follows names a procedure by adding a fifth element to the procedure data list, which is the procedures name:

```
'(procedure (x) ((+ x 5)) <procedure-env> foo)
```

This naming occurs when the lambda is bound to a variable for the first time. So if we are setting a variable to some value and that value happens to be an unnamed lambda, then we will name this lambda with the variable name that we are setting. To do this the following function was defined and a call to it was added to `eval-assignment` and `eval-definition`. The code follows:

```
;; this is added for naming procedures.
;;  if the value is a lambda, then we will set the name of
;;  of the lambda (the 5th item in the list) if it is not
;;  already set to some value.
(define (set-procedure-name-if-necessary var proc)
        ;; if it is a procedure and its not
        ;; named, then set the name.
        (if (and (compound-procedure? proc)
                 (not (procedure-named? proc))
            )
            (set-cdr! (cdddr proc) (list var))
        )
)
```

**Making `my-apply` print out trace information:** Here, if we have a compound procedure, before we call the procedure, we check to see if we have a traced procedure. If it is traced then we print out procedure call information. We then call the procedure in the same way that it was called in the original code. The return value is saved in `retval`. Then, if we this procedure is traced then we print out return information. Finally we return the `retval`.

```
(define (my-apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)

;;; BEGIN: added for problem 2.
          ;; if it is a traced function
          ;; print out entering info.
          (cond ((traced-procedure? procedure)
                 (display "Entering Procedure: ")
                 (if (procedure-named? procedure)
                     (display (procedure-name procedure))
                     (user-print procedure)
                 )
                 (newline)
                 (display "Arguments:\n")
                 (vars-vals-print
                    (procedure-parameters procedure)
                    arguments
                 )
                )
          )

          ;; invoke procedure.  This time though, we will will
          ;;  store the return value in variable 'retval'.
          ;; we have to save the return value because we
          ;;  want to print it out if this is a traced
          ;;  procedure.
          (let ((retval

;;; BEGIN: original code to call a function.
                   (eval-sequence
                     (procedure-body procedure)
                     (extend-environment
                       (procedure-parameters procedure)
                       arguments
                       (procedure-environment procedure)))
                   )
```

```
;;; END: of original code to call function.
                   ;; now the function has returned and we have
                   ;; retval.
                )

                ;; if it is a traced function
                ;; print exit info.
                (cond ((traced-procedure? procedure)
                       (display "Leaving Procedure: ")
                       (if (procedure-named? procedure)
                           (display (procedure-name procedure))
                           (user-print procedure)
                       )
                       (newline)
                       (display "\tReturning: ")
                       (user-print retval)
                       (newline)
                      )
                )

                ;; return retval
                retval
          )
;;; END: added for problem 2 and 3.

        )
        (else
         (error
          "Unknown procedure type -- APPLY" procedure))))
```

**Adding `my-trace` and `my-untrace`:** Here we just add `my-trace` and `my-untrace` to our primitives with the following primitive functions:

```
;; here are my-trace and my-untrace for problem 2.
(define (my-trace p)
  (set-car! p 'traced-procedure)
  p    ; return p
)

(define (my-untrace p)
  (set-car! p 'procedure)
  p    ; return p
)
```

**Output:**

```
;;; M-Eval input:
```

```
(trace factorial)

;;; M-Eval value:
(compound-procedure (n) ((if (= 0 n) 1 (* n (fact...

;;; M-Eval input:
(factorial 3)
Entering Procedure: factorial
Arguments:
        n       ->      3
Entering Procedure: factorial
Arguments:
        n       ->      2
Entering Procedure: factorial
Arguments:
        n       ->      1
Entering Procedure: factorial
Arguments:
        n       ->      0
Leaving Procedure: factorial
        Returning: 1
Leaving Procedure: factorial
        Returning: 1
Leaving Procedure: factorial
        Returning: 2
Leaving Procedure: factorial
        Returning: 6

;;; M-Eval value:
6
```

3. Also add a second, more verbose, trace facility that prints out the environment in a nice way after entering a function, in addition to the information from `my-trace`. The printout should include the names and values of each variable in each frame. Indicate the different frames as well. This verbose trace facility should be invoked using `(enviro-trace factorial)`. Demonstrate both `my-trace` and `enviro-trace` working on the recursive functions `factorial`, `append`, and `map`.

   **Solution:** Once `my-trace` is implemented, adding `enviro-trace` is rather simple. It involved writing a function to print out the environment without getting into an infinite loop on the `<procedure-env>` portion of procedures that are stored in variables.

   **Printing out the environment:** The trick here is that when ever a *value* is printed out, `user-print` must be used.

```
;;; for problem 2 and 3.
;;    This prints out a variable
;;    and value pair in a nice way.
(define (var-val-print var val)
   (display "\t")
   (display var)
   (display "\t->\t")
   (user-print val)
   (display "\n")
)

;;; for problem 2 and 3.
;;    This prints out a lists of variables
;;    and values in a nice way.
;;       - calls var-val-print
(define (vars-vals-print vars vals)
   (map var-val-print vars vals)
)

;;; END: problem 2.


;;; for problem 3.  Print the environment.
;;    the environment is a list of list.
;;    we will print out the first list.
;;    then recursively print out the rest.
;;        - calls vars-vals-print
(define (env-print env)
   (cond ((null? env) '())   ; do nothing on empty list.
         (else

            (display "Environment[")
            (display (length env))
            (display "]:\n")
            ;; first print the first environment.
            (let ((frame (first-frame env)))
                 (vars-vals-print
                    (frame-variables frame)
                    (frame-values frame)
                 )
            )

            ;; print the rest recursively.
            (env-print (enclosing-environment env))
```

```
          )
      )
    )
```

**Marking a procedure as enviro-traced:** We will, as before, change `'procedure` to `'enviro-traced-procedure` and add the following function:

```
(define (traced-procedure? p)
  (or (tagged-list? p 'traced-procedure)
      (enviro-traced-procedure? p)
  )
)

(define (enviro-traced-procedure? p)
  (tagged-list? p 'enviro-traced-procedure)
)
```

**Changing `my-apply`:** All that is left is to add a bit to `my-apply` to to call `env-print` before an enviro-traced procedure is called.

**The `append` and `map` functions:**

```
;; map:  maps function f on list l
(define (map f l)
  (if (null? l)
      '()
      (cons (f (car l)) (map f (cdr l)))
  )
)

;; append: append t onto the end of s.
(define (append s t)
  ( if (null? s)
       t
       (cons (car s) (append (cdr s) t))
  )
)
```

**Output:** (for brevity, repetitive portions of the output are left out)

```
;;; M-Eval input:
(enviro-trace append)

;;; M-Eval value:
(compound-procedure (s t) ((if (null? s)...

;;; M-Eval input:
(append '(1 2) '(10 11 12))
```

```
Entering Procedure: append
Arguments:
        s       ->      (1 2)
        t       ->      (10 11 12)
Environment[1]:
        append  ->      (compound-proced...
        map     ->      (compound-proced...
        factorial       ->      (compoun...
        false   ->      #f
        true    ->      #t
        car     ->      (primitive #[<closure> car])
        cdr     ->      (primitive #[<closure> cdr])
        ...
        =       ->      (primitive #[<closure> full=?])
        null?   ->      (primitive #[<closure> null?])
        env     ->      (primitive #[<cl...
        enviro-trace    ->      (primiti...
Entering Procedure: append
Arguments:
        s       ->      (2)
        t       ->      (10 11 12)
Environment[1]:
        append  ->      (compound-proced...
        map     ->      (compound-proced...
        factorial       ->      (compoun...
        false   ->      #f
        true    ->      #t
        car     ->      (primitive #[<closure> car])
        cdr     ->      (primitive #[<closure> cdr])
        cons    ->      (primitive #[<closure> cons])
        ...
        env     ->      (primitive #[<cl...
        enviro-trace    ->      (primiti...
Entering Procedure: append
Arguments:
        s       ->      ()
        t       ->      (10 11 12)
Environment[1]:
        append  ->      (compound-proced...
        map     ->      (compound-proced...
        factorial       ->      (compoun...
        false   ->      #f
        true    ->      #t
        car     ->      (primitive #[<closure> car])
```

```
        cdr     ->      (primitive #[<closure> cdr])
        ...
        null?   ->      (primitive #[<closure> null?])
        env     ->      (primitive #[<cl...
        enviro-trace    ->      (primiti...
Leaving Procedure: append
        Returning: (10 11 12)
Leaving Procedure: append
        Returning: (2 10 11 12)
Leaving Procedure: append
        Returning: (1 2 10 11 12)

;;; M-Eval value:
(1 2 10 11 12)

;;; M-Eval input:
(enviro-trace map)

;;; M-Eval value:
(compound-procedure (f l) ((if (null? ...

;;; M-Eval input:
(map factorial '(0 1))
Entering Procedure: map
Arguments:
        f       ->      (compound-proc...
        l       ->      (0 1)
Environment[1]:
        append  ->      (compound-proc...
        map     ->      (compound-proc...
        factorial       ->      (compo...
        false   ->      #f
        true    ->      #t
        car     ->      (primitive #[<...
        ...
        =       ->      (primitive #[<...
        null?   ->      (primitive #[<...
        env     ->      (primitive #[<...
        enviro-trace    ->      (primi...
Entering Procedure: factorial
Arguments:
        n       ->      0
Leaving Procedure: factorial
        Returning: 1
```

```
Entering Procedure: map
Arguments:
        f       ->      (compound-proc...
        l       ->      (1)
Environment[1]:
        append  ->      (compound-proc...
        map     ->      (compound-proc...
        factorial       ->      (compo...
        false   ->      #f
        true    ->      #t
        car     ->      (primitive #[<...
        cdr     ->      (primitive #[<...
        ...
        env     ->      (primitive #[<...
        enviro-trace    ->      (primi...
Entering Procedure: factorial
Arguments:
        n       ->      1
Entering Procedure: factorial
Arguments:
        n       ->      0
Leaving Procedure: factorial
        Returning: 1
Leaving Procedure: factorial
        Returning: 1
Entering Procedure: map
Arguments:
        f       ->      (compound-proc...
        l       ->      ()
Environment[1]:
        append  ->      (compound-proc...
        map     ->      (compound-proc...
        factorial       ->      (compo...
        false   ->      #f
        true    ->      #t
        car     ->      (primitive #[<...
        ...
        null?   ->      (primitive #[<...
        env     ->      (primitive #[<...
        enviro-trace    ->      (primi...
Leaving Procedure: map
        Returning: ()
Leaving Procedure: map
        Returning: (1)
```

```
Leaving Procedure: map
        Returning: (1 1)

;;; M-Eval value:
(1 1)
```

4. Exercise 4.4 (page 374)

   Add special forms for short-circuiting `or` and `and` as well as derived, source-to-source, transformations. For brevity we will give a short-circuiting `and` and a special form for `or`[3].

   **`and` special form solution:**

```
(define (special-and? exp)
   (tagged-list? exp 'special-and)
)

;; this us used for getting all the expressions for
;; a special-and or special-or.  returns a list.
(define (special-bool-exps exp)
   (cdr exp)
)

;; this does a special-and
(define (eval-special-and exp env)
   (letrec ((do-and
               (lambda (exps)
                  (if (null? exps)
                      true
                      (and (my-eval (car exps) env)
                           (do-and (cdr exps))
                      )
                  )
               )
            )
           )
           (do-and (special-bool-exps exp))
   )
)
```

   **`or` source-to-source solution:**

```
define (derived-or? exp)
   (tagged-list? exp 'derived-or)
```

[3]Both are implemented in `hw2.s`

```
)

;; these are the same, lets recycle.
(define derived-bool-exps special-bool-exps)

(define (orexps->if exps)
   (if (null? exps)
       'false
       (make-if (car exps) 'true (orexps->if (cdr exps)))
   )
)

;; this makes a huge nested if
(define (or->if exp)
   (orexps->if (derived-bool-exps exp))
)
```

**Change to** `my-eval`

```
(define (my-eval exp env)
  (cond ((self-evaluating? exp) exp)
        ...
        ((special-and? exp) (eval-special-and exp env))
        ((derived-or? exp) (my-eval (or->if exp) env))
        ...
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

**Output:**

```
;;; M-Eval input:
(derived-or false true (/ 1 0))

;;; M-Eval value:
#t

;;; M-Eval input:
(special-and true true true false (/ 1 0))

;;; M-Eval value:
#f

;;; M-Eval input:
(derived-or false)
```

```
;;; M-Eval value:
#f

;;; M-Eval input:
(derived-or)

;;; M-Eval value:
#f

;;; M-Eval input:
(special-and)

;;; M-Eval value:
#t
```

5. Exercise 4.6 (page 375)

   Implement `let` as a source-to-source transformation to `lambda`

   **Transformation solution:**

```
(define (make-combination func args)
   (cons func args)
)


(define (let? exp)
   (tagged-list? exp 'let)
)

;; the body is the third+ expression(s).
(define (let-body exp)
   (cddr exp)
)

;; this gets out the definitions into a list:
;;   ((var1 exp1) (var2 exp2) ...)
(define (let-defs-raw exp)
   (cadr exp)
)

;; this gets out the definitions from a paired list:
;;   ((var1 exp1) (var2 exp2) ...)
;; and puts them into a a pair of list:
;; ((var1 var2 ...) (exp1 exp2 ...)
(define (let-defs-unpair raw)
```

```
    (if (null? raw)
        '( () () )  ; return two empty lists.
        (let* ((top      (car raw))
               (new-var  (car top))
               (new-val  (cadr top))
               (the-rest (let-defs-unpair (cdr raw)))
               (vars (car the-rest))
               (vals (cadr the-rest))
              )
              (list (cons new-var vars) (cons new-val vals))
        )
    )
)

;; this gets out the definitions into a list:
;; ((var1 var2 ...) (exp1 exp2 ...)
(define (let-defs exp)
   (let-defs-unpair (let-defs-raw exp))
)

;; turn this let into an application of a lambda
(define (let->combination exp)
   (let* ((vars-exps (let-defs exp))
          (vars (car vars-exps))
          (exps (cadr vars-exps))
         )
         (make-combination
            (make-lambda vars (let-body exp))
            exps
         )
   )
)
```

**New** `my-eval`**:**

```
(define (my-eval exp env)
  (cond ((self-evaluating? exp) exp)
        ...
        ((let? exp) (my-eval (let->combination exp) env))
        ...
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

**Output:**

```
;;; M-Eval input:
```

```
(let ((x 5) (y 10)) (+ x y))

;;; M-Eval value:
15

;;; M-Eval input:
(let () 1 2 3)

;;; M-Eval value:
3
```

6. Exercise 4.7 (page 375)

   Define `let*` as a source-to-source transformation to a chain of nested `lets` if it is possible.

   **Is is possible?** Yes.

   **Transformation solution:**

```
;; this makes a let for my-eval.
;;    vars-exps: ((var1 val1) (var2 val2) ...)
;;    body:  ((exp1) ...)

(define (make-let vars-exps body)
   (append (list 'let vars-exps) body)
)


(define (let*? env)
   (tagged-list? env 'let*)
)

;; let* has the same syntax as let,
;; so we will use let's accessor function: let-defs-raw and let-body
(define (let*->nested-lets exp)
   (raw-body-let*->nested-lets (let-defs-raw exp) (let-body exp))
)

(define (raw-body-let*->nested-lets raw body)
   (if (null? raw)
       (make-begin body)
       (make-let (list (car raw))
                 (list (raw-body-let*->nested-lets (cdr raw) body))
       )
   )
)
```

**New** `my-eval`**:**

```
(define (my-eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ...
        ((let*? exp) (my-eval (let*->nested-lets exp) env))
        ...
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

**Output:**

```
;;; M-Eval input:
(let* ((x 5) (y (+ x 10)) (z (+ y 100))) z)

;;; M-Eval value:
115
```

The entire Homework 2 solution:

```
;;; CSE 341 Homework 2 Solutions
;;;
;;; Jason Hartline


;;;; METACIRCULAR EVALUATOR FROM CHAPTER 4 (SECTIONS 4.1.1-4.1.4) of
;;;; STRUCTURE AND INTERPRETATION OF COMPUTER PROGRAMS

;;;;Matches code in ch4.scm

;;;;This file can be loaded into Scheme as a whole.
;;;;Then you can initialize and start the evaluator by evaluating
;;;; the two commented-out lines at the end of the file (setting up the
;;;; global environment and starting the driver loop).

;;;from section 4.1.4 -- must precede def of metacircular apply
(define apply-in-underlying-scheme apply)

;;;SECTION 4.1.1

(define (my-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))

;;; BEGIN: added for problem 4.
        ((special-or? exp) (eval-special-or exp env))
        ((special-and? exp) (eval-special-and exp env))
        ((derived-or? exp) (my-eval (or->if exp) env))
        ((derived-and? exp) (my-eval (and->if exp) env))

;; real 'and' is 'special'
        ((and? exp) (eval-special-and exp env))

;; real 'or' is 'derived'
        ((or? exp) (my-eval (or->if exp) env))
;;; END: added for problem 4.

;;; BEGIN : added for problem 5.
        ((let? exp) (my-eval (let->combination exp) env))
```

```
;;; END: added for problem 5.

;;; BEGIN: added for problem 6.
        ((let*? exp) (my-eval (let*->nested-lets exp) env))
;;; END: added for problem 6.

        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (my-eval (cond->if exp) env))
        ((application? exp)
         (my-apply (my-eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))

;;; added for problem 2: a "traced-procedure"

(define (my-apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)

;;; BEGIN: added for problem 2 and 3.

          ;; if it is a traced function
          ;; print out entering info.
          (cond ((traced-procedure? procedure)
                 (display "Entering Procedure: ")
                 (if (procedure-named? procedure)
                     (display (procedure-name procedure))
                     (user-print procedure)
                 )
                 (newline)
                 (display "Arguments:\n")
                 (vars-vals-print
     (procedure-parameters procedure)
     arguments
 )

                 ;; if we have an enviro-traced procedure
```

```
                ;; then print out the environment.
                (cond ((enviro-traced-procedure? procedure)
                       (env-print
   (procedure-environment procedure)
)
                      )
                )
               )
         )



         ;; invoke procedure.  This time though, we will will
         ;;  store the return value in variable 'retval'.
         ;; we have to save the return value because we
         ;;  want to print it out if this is a traced
         ;;  procedure.
         (let ((retval

;;; BEGIN: origional code to call a function.
                  (eval-sequence
                    (procedure-body procedure)
                    (extend-environment
                      (procedure-parameters procedure)
                      arguments
                      (procedure-environment procedure)))
                  )
;;; END: of origional code to call function.
                  ;; now the function has returned and we have
                  ;; retval.
               )

               ;; if it is a traced function
               ;; print exit info.
               (cond ((traced-procedure? procedure)
                      (display "Leaving Procedure: ")
                      (if (procedure-named? procedure)
                          (display (procedure-name procedure))
                          (user-print procedure)
                      )
                      (newline)
                      (display "\tReturning: ")
                      (user-print retval)
                      (newline)
```

```
                       )
                 )

                 ;; return retval
                 retval
           )
;;; END: added for problem 2 and 3.

        )
        (else
         (error
          "Unknown procedure type -- APPLY" procedure))))


(define (list-of-values exps env)
  (if (no-operands? exps)
      '()
      (cons (my-eval (first-operand exps) env)
            (list-of-values (rest-operands exps) env))))

(define (eval-if exp env)
  (if (true? (my-eval (if-predicate exp) env))
      (my-eval (if-consequent exp) env)
      (my-eval (if-alternative exp) env)))

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

;;; BEGIN: added for problem 2

;; this is added for naming procedures.
;;  if the value is a lamnda, then we will set the name of
;;  of the lambda (the 5th item in the list) if it is not
;;  already set to some value.
(define (set-procedure-name-if-necessary var proc)
        ;; if it is a procedure and its not
        ;; named, then set the name.
        (if (and (compound-procedure? proc)
         (not (procedure-named? proc))
    )
          (set-cdr! (cdddr proc) (list var))
        )
```

```
)

;;; END: added for problem 2

;;; BEGIN: modified for problem 2

;; this originally took care of
;; setting a variable with set!
;; we have added the bit to set
;; the procedure name if the value
;; is a lambda.
(define (eval-assignment exp env)
  (let ((var (assignment-variable exp))
        (val (my-eval (assignment-value exp) env))
       )
       (set-procedure-name-if-necessary var val)
       (set-variable-value! var val env )
       'ok
  )
)

;; this originally took care of
;; setting a variable with define
;; we have added the bit to set
;; the procedure name if the value
;; is a lambda.
(define (eval-definition exp env)
  (let ((var (definition-variable exp))
        (val (my-eval (definition-value exp) env))
       )
       (set-procedure-name-if-necessary var val)
       (define-variable! var val env )
       'ok
  )
)

;;; END: modified for problem 2



;;;SECTION 4.1.2

(define (self-evaluating? exp)
  (cond ((number? exp) true)
```

```
        ((string? exp) true)
        (else false)))

(define (quoted? exp)
  (tagged-list? exp 'quote))

(define (text-of-quotation exp) (cadr exp))

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

(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))))

(define (lambda? exp) (tagged-list? exp 'lambda))

(define (lambda-parameters exp) (cadr exp))
(define (lambda-body exp) (cddr exp))

(define (make-lambda parameters body)
  (cons 'lambda (cons parameters 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 predicate consequent alternative)
  (list 'if predicate consequent alternative))


(define (begin? exp) (tagged-list? exp 'begin))

(define (begin-actions exp) (cdr 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 seq) (cons 'begin seq))


(define (application? exp) (pair? exp))
(define (operator exp) (car exp))
(define (operands exp) (cdr exp))

(define (no-operands? ops) (null? ops))
(define (first-operand ops) (car ops))
(define (rest-operands ops) (cdr ops))


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

```
(define (cond-clauses exp) (cdr exp))

(define (cond-else-clause? clause)
  (eq? (cond-predicate clause) 'else))

(define (cond-predicate clause) (car clause))

(define (cond-actions clause) (cdr clause))

(define (cond->if exp)
  (expand-clauses (cond-clauses exp)))

(define (expand-clauses clauses)
  (if (null? clauses)
      'false                          ; no else clause
      (let ((first (car clauses))
            (rest (cdr clauses)))
        (if (cond-else-clause? first)
            (if (null? rest)
                (sequence->exp (cond-actions first))
                (error "ELSE clause isn't last -- COND->IF"
                       clauses))
            (make-if (cond-predicate first)
                     (sequence->exp (cond-actions first))
                     (expand-clauses rest))))))

;;;SECTION 4.1.3

(define (true? x)
  (not (eq? x false)))

(define (false? x)
  (eq? x false))


(define (make-procedure parameters body env)
  (list 'procedure parameters body env))

;;; BEGIN: modified for problem 2.
(define (compound-procedure? p)
  (or (tagged-list? p 'procedure) (traced-procedure? p))
)
;;; END: modified for problem 2.
```

```
;;; BEGIN: added for problem 2.
(define (traced-procedure? p)
  (or (tagged-list? p 'traced-procedure)
      (enviro-traced-procedure? p)
  )
)

(define (enviro-traced-procedure? p)
  (tagged-list? p 'enviro-traced-procedure)
)

;; here are my-trace and my-untrace for problem 2.
(define (my-trace p)
  (set-car! p 'traced-procedure)
  p    ; return p
)

(define (my-untrace p)
  (set-car! p 'procedure)
  p    ; return p
)

;;; END: added for problem 2.


;;; BEGIN: added for problem 3.

;; this marks a procedure as enviro-traced
;;   note: we will use my-untrace to enviro-untrace
(define (my-enviro-trace p)
  (set-car! p 'enviro-traced-procedure)
  p    ; return p
)

;;; END: here is my-enviro-trace for problem 3.





(define (procedure-parameters p) (cadr p))
(define (procedure-body p) (caddr p))
(define (procedure-environment p) (cadddr p))
```

```
(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 arguments supplied" vars vals)
          (error "Too few arguments 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" 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))
            (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))))

;;;SECTION 4.1.4

(define (setup-environment)
  (let ((initial-env
         (extend-environment (primitive-procedure-names)
                             (primitive-procedure-objects)
                             the-empty-environment)))
    (define-variable! 'true true initial-env)
    (define-variable! 'false false initial-env)
    initial-env))

;[do later] (define the-global-environment (setup-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?)

;;; BEGIN: added to make life easier.
        (list 'list list)
;;; END: added to make life easier.

;;; BEGIN: added for problem 1
        (list '+ +)
        (list '* *)
        (list '- -)
        (list '/ /)
;;; END: added for problem 1

;;; BEGIN: added for problem 2
        (list 'trace my-trace)
        (list 'untrace my-untrace)
        (list '= =)
;;; END: added for problem 2

;;; BEGIN: added for problem 3
        (list 'null? null?)
        ;; this is helpful
        (list 'env (lambda () (env-print the-global-environment)))
        (list 'enviro-trace my-enviro-trace)
;;; END: added for problem 3

;;      more primitives
        ))

(define (primitive-procedure-names)
  (map car
       primitive-procedures))

(define (primitive-procedure-objects)
  (map (lambda (proc) (list 'primitive (cadr proc)))
       primitive-procedures))

;[moved to start of file] (define apply-in-underlying-scheme apply)

(define (apply-primitive-procedure proc args)
  (apply-in-underlying-scheme
   (primitive-implementation proc) args))
```

```
(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 (my-eval input the-global-environment)))
      (announce-output output-prompt)
      (user-print output)))
  (driver-loop))

(define (prompt-for-input string)
  (newline) (newline) (display string) (newline))

(define (announce-output string)
  (newline) (display string) (newline))

(define (user-print object)
  (if (compound-procedure? object)
      (display (list 'compound-procedure
                     (procedure-parameters object)
                     (procedure-body object)
                     '<procedure-env>
                     (procedure-name object)))
      (display object)))


;;; BEGIN: for problem 2.

;; this tries to get a procedure name.
;;   We have modified the data structure for a
;;   lambda by adding an optional 5th field.
;;   This fifth field contains the name of the
;;   lambda.  (this name gets set when a variable
;;   is set or defined.
(define (procedure-name p)
   (let ((last (cddddr p)))
      ;; if there is no name ...
      (if (null? last)
         "unnamed"

         ;; if there is a name ...
         (car last)
```

```
      )
   )
)

;;; also for problem 2.
(define (procedure-named? p)
   (and (compound-procedure? p) (not (null? (cddddr p))))
)


;;; for problem 2 and 3.
;;    This prints out a variable
;;    and value pair in a nice way.
(define (var-val-print var val)
   (display "\t")
   (display var)
   (display "\t->\t")
   (user-print val)
   (display "\n")
)

;;; for problem 2 and 3.
;;    This prints out a lists of variables
;;    and values in a nice way.
;;       - calls var-val-print
(define (vars-vals-print vars vals)
   (map var-val-print vars vals)
)

;;; END: problem 2.


;;; for problem 3.  Print the environment.
;;    the environment is a list of list.
;;    we will print out the first list.
;;    then recursively print out the rest.
;;        - calls vars-vals-print
(define (env-print env)
   (cond ((null? env) '())   ; do nothing on empty list.
         (else

            (display "Environment[")
            (display (length env))
            (display "]:\n")
```

```
            ;; first print the first environment.
            (let ((frame (first-frame env)))
                 (vars-vals-print
     (frame-variables frame)
     (frame-values frame)
 )
            )

            ;; print the rest recursively.
            (env-print (enclosing-environment env))
         )
   )
)



;;; this is for problem 4 (exercise 4.4)
;;; this makes a special form for or and and.
(define (special-or? exp)
   (tagged-list? exp 'special-or)
)

(define (special-and? exp)
   (tagged-list? exp 'special-and)
)

;; this us used for getting all the expressions for
;; a special-and or special-or.  returns a list.
(define (special-bool-exps exp)
   (cdr exp)
)

;; this does a special-or
(define (eval-special-or exp env)
   (display "doing or\n")
   (letrec ((do-or
               (lambda (exps)
                  (if (null? exps)
                      false
                      (or (my-eval (car exps) env)
          (do-or (cdr exps))
      )
                  )
               )
```

```
             )
            )
            (do-or (special-bool-exps exp))
   )
)

;; this does a special-and
(define (eval-special-and exp env)
   (letrec ((do-and
                (lambda (exps)
                   (if (null? exps)
                       true
                       (and (my-eval (car exps) env)
            (do-and (cdr exps))
      )
                   )
                )
             )
            )
            (do-and (special-bool-exps exp))
   )
)



;;; this makes a source to source transformation
;;; and derives or/and from if.

(define (derived-and? exp)
   (tagged-list? exp 'derived-and)
)

(define (derived-or? exp)
   (tagged-list? exp 'derived-or)
)

;; these are the same, lets recycle.
(define derived-bool-exps special-bool-exps)

(define (orexps->if exps)
   (if (null? exps)
       'false
       (make-if (car exps) 'true (orexps->if (cdr exps)))
   )
```

```
)

;; this makes a huge nested if
(define (and->if exp)
   (andexps->if (derived-bool-exps exp))
)

(define (andexps->if exps)
   (if (null? exps)
       'true
       (make-if (car exps) (orexps->if (cdr exps)) 'false)
   )
)

;; this makes a huge nested if
(define (or->if exp)
   (orexps->if (derived-bool-exps exp))
)

;;; now lets make 'or' be the derived-or
;;; and 'and' be the special-and
(define (or? exp)
   (tagged-list? exp 'or)
)

(define (and? exp)
   (tagged-list? exp 'and)
)


;;; BEGIN: for problem 5

(define (make-combination func args)
   (cons func args)
)


(define (let? exp)
   (tagged-list? exp 'let)
)

;; the body is the third+ expression(s).
(define (let-body exp)
   (cddr exp)
```

```
)

;; this gets out the definitions into a list:
;;   ((var1 exp1) (var2 exp2) ...)
(define (let-defs-raw exp)
   (cadr exp)
)

;; this gets out the definitions from a paired list:
;;   ((var1 exp1) (var2 exp2) ...)
;; and puts them into a a pair of list:
;; ((var1 var2 ...) (exp1 exp2 ...)
(define (let-defs-unpair raw)
   (if (null? raw)
       '( () () )  ; return two empty lists.
       (let* ((top      (car raw))
              (new-var  (car top))
              (new-val  (cadr top))
              (the-rest (let-defs-unpair (cdr raw)))
              (vars (car the-rest))
              (vals (cadr the-rest))
             )
             (list (cons new-var vars) (cons new-val vals))
       )
   )
)

;; this gets out the definitions into a list:
;; ((var1 var2 ...) (exp1 exp2 ...)
(define (let-defs exp)
   (let-defs-unpair (let-defs-raw exp))
)

;; turn this let into an application of a lambda
(define (let->combination exp)
   (let* ((vars-exps (let-defs exp))
          (vars (car vars-exps))
          (exps (cadr vars-exps))
         )
         (make-combination
            (make-lambda vars (let-body exp))
            exps
         )
   )
```

```
)

;;; END: for problem 5


;;; BEGIN: for problem 6

;; this makes a let for my-eval.
;;    vars-exps: ((var1 val1) (var2 val2) ...)
;;    body:  ((exp1) ...)

(define (make-let vars-exps body)
   (append (list 'let vars-exps) body)
)


(define (let*? env)
   (tagged-list? env 'let*)
)

;; let* has the same syntax as let,
;; so we will use let's accessor functions:
;; let-defs-raw and let-body.

(define (let*->nested-lets exp)
   (raw-body-let*->nested-lets
     (let-defs-raw exp)
     (let-body exp)
   )
)

(define (raw-body-let*->nested-lets raw body)
   (if (null? raw)
       (make-begin body)
       (make-let
          (list (car raw))
          (list (raw-body-let*->nested-lets (cdr raw) body))
       )
   )
)


;;; END: for problem 6
```

```
(define the-global-environment (setup-environment))

(define (start . x)
   (display "restarting\n")
   (set! the-global-environment (setup-environment))
   (driver-loop)
)

(define (reload)
   (load "my-eval.s")
)


'METACIRCULAR-EVALUATOR-LOADED
```