Exercise 4.31.

The approach taken in this section is somewhat unpleasant, because it makes an incompatible change to Scheme. It might be nicer to implement lazy evaluation as an *upward-compatible extension*, that is, so that ordinary Scheme programs will work as before. We can do this by extending the syntax of procedure declarations to let the user control whether or not arguments are to be delayed. While we're at it, we may as well also give the user the choice between delaying with and without memoization. For example, the definition

```
(define (f a (b lazy) c (d lazy-memo))
  ...)
```

would define `f` to be a procedure of four arguments, where the first and third arguments are evaluated when the procedure is called, the second argument is delayed, and the fourth argument is both delayed and memoized. Thus, ordinary procedure definitions will produce the same behavior as ordinary Scheme, while adding the `lazy-memo` declaration to each parameter of every compound procedure will produce the behavior of the lazy evaluator defined in this section. Design and implement the changes required to produce such an extension to Scheme. You will have to implement new syntax procedures to handle the new syntax for `define`. You must also arrange for `eval` or `apply` to determine when arguments are to be delayed, and to force or delay arguments accordingly, and you must arrange for forcing to memoize or not, as appropriate.

Answer.

Modifying the evaluator
Extending the evaluator upward-compatibly in this way requires additional syntax procedures to determine the `lazy` and `lazy-memo` declaration in procedure definition.

```
(define (lazy? var)
  (if (pair? var)
      (eq? (cadr var) 'lazy)
      false))

(define (lazy-memo? var)
  (if (pair? var)
      (eq? (cadr var) 'lazy-memo)
      false))
```

`Eval` remains the same because the new evaluator involves both applicative and normal order evaluation and `actual-value` is devised to handle both cases. The essentials in extending the evaluator upward-compatibly is in the handling of procedure applications in `apply` compared to the lazy evaluator. For primitive procedures, we still evaluate all the arguments before applying the primitive; for compound procedures, we either evaluate or delay an argument before applying the procedure, according to the programmer's choice.

```
(define (apply procedure arguments env)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure
          procedure
          (list-of-arg-values arguments env)))
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment
           (procedure-parameters procedure)
           (list-of-accord-args (procedure-parameters procedure)     ; changed
                                arguments
                                env)
```

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

The procedure `list-of-accord-args` performs selective evaluation on procedure arguemnts at the programmer's preference.

```
(define (list-of-accord-args pars args env)
  (if (no-operands? args)
      '()
      (cons (accord-eval (first-operand pars)
                         (first-operand args)
                         env)
            (list-of-accord-args (rest-operands pars)
                                 (rest-operands args)
                                 env))))

(define (accord-eval par arg env)
  (cond ((variable? par)
         (eval arg env))
        ((lazy? par)
         (delay-it arg env))
        ((lazy-memo? par)
         (memo-delay-it arg env))
        (else
         (error "Unknown parameter type -- DEFINE" par))))
```

Finally, we also change the prompt to indicate that this is the upward-compatible evaluator:

```
(define input-prompt ";;; UC-Eval input:")
(define output-prompt ";;; UC-Eval value:")
```

With these changes made, we can start the evaluator and test it. The successful evaluation of the `try` expression discussed in section 4.2.1 indicates that the interpreter is preforming upward-compatible evaluation:

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

(driver-loop)

;;; UC-Eval input:
(define (try a b)
  (if (= a 0) 1 b))
;;; UC-Eval value:
ok

;;; UC-Eval input:
(try 0 (/ 1 0))
;Division by zero signalled by /.
;To continue, call RESTART with an option number:
; (RESTART 1) => Return to read-eval-print level 1.

(RESTART 1)
;Abort!

(driver-loop)

;;; UC-Eval input:
(define (try a (b lazy))
  (if (= a 0) 1 b))
;;; UC-Eval value:
```

```
ok

;;; UC-Eval input:
(try 0 (/ 1 0))
;;; UC-Eval value:
1

;;; UC-Eval input:
(define (try a (b lazy-memo))
  (if (= a 0) 1 b))
;;; UC-Eval value:
ok

;;; UC-Eval input:
(try 0 (/ 1 0))
;;; UC-Eval value:
1
```

Representing thunks
The extended evaluator must manage to create thunks correspondingly to meet the programmer's request and force them later. For the ordinary delayed request, it follows the way in the text by handling thunks without memoization; for requests of both delayed and memoized, thunks are turned into evaluated ones when forced

```
(define (delay-it exp env)
  (list 'thunk exp env))

(define (thunk? obj)
  (tagged-list? obj 'thunk))

(define (thunk-exp thunk) (cadr thunk))

(define (thunk-env thunk) (caddr thunk))

(define (memo-delay-it exp env)
  (list 'memo-thunk exp env))

(define (memo-thunk? obj)
  (tagged-list? obj 'memo-thunk))

(define (evaluated-thunk? obj)
  (tagged-list? obj 'evaluated-thunk))

(define (thunk-value evaluated-thunk) (cadr evaluated-thunk))

(define (force-it obj)
  (cond ((thunk? obj)
         (actual-value (thunk-exp obj) (thunk-env obj)))
        ((memo-thunk? obj)
         (let ((result (actual-value
                         (thunk-exp obj)
                         (thunk-env obj))))
           (set-car! obj 'evaluated-thunk)
           (set-car! (cdr obj) result)
           (set-cdr! (cdr obj) '())
           result))
        ((evaluated-thunk? obj)
         (thunk-value obj))
        (else obj)))
```