Exercise 3.30

The value in an extend-env is an expressed value, so the value of saved-val in apply-env is an expressed value.

Exercise 3.31

The grammar is changed to satisfy the specification in exercise 3.21. The procedure constructor now accepts a list of identifiers. Where there are calls to procedure, we make plural the names of the variables for our sake, and those places are commented. In value-of, proc-exp calls procedure and returns an expressed value. Hence extend-env remains unchanged. But extend-env-rec must now take a list of symbols. And apply-procedure is defined in terms of extend-env*.

```
;; the-grammar
(expression
 ("proc" "(" (separated-list identifier ",") ")" expression)
 proc-exp)

(expression
 ("(" expression (arbno expression) ")")
 call-exp)

(expression
 ("letrec"
  identifier "(" (separated-list identifier ",") ")" "="
  expression
  "in" expression)
 letrec-exp)

;; value-of
(proc-exp (vars body)                              ; var -> vars
          (proc-val (procedure vars body env))) ; var -> vars
(call-exp (rator rands)
          (let ((proc (expval->proc (value-of rator env)))
                (args (map (lambda (rand) (value-of rand env))
                           rands)))
            (apply-procedure proc args)))
(letrec-exp (p-name b-vars p-body letrec-body) ; b-var -> b-vars
            (value-of letrec-body
                      (extend-env-rec p-name
                                      b-vars   ; b-var -> b-vars
                                      p-body
                                      env)))

;; apply-procedure : Proc * Listof(ExpVal) -> ExpVal
(define apply-procedure
  (lambda (proc1 args)
    (cases proc proc1
      (procedure (vars body saved-env)
                 (value-of body (extend-env* vars args saved-env))

;; proc? : SchemeVal -> Bool
;; procedure : Var * Exp * Env -> Proc
(define-datatype proc proc?
  (procedure
   (bvars (list-of symbol?))
   (body expression?)
   (env environment?)))

(define-datatype environment environment?
  (empty-env)
  (extend-env
   (bvar symbol?)
   (bval expval?)
   (saved-env environment?))
  (extend-env-rec
   (id symbol?)
   (bvars (list-of symbol?))
   (body expression?)
   (saved-env environment?)))

;; apply-env : Env * Var -> ExpVal
(define apply-env
  (lambda (env search-sym)
    (cases environment env
      (empty-env ()
                 (eopl:error 'apply-env
                             "No binding for ~s"
                             search-sym))
      (extend-env (var val saved-env)
                  (if (eqv? search-sym var)
                      val
                      (apply-env saved-env search-sym)))
      (extend-env-rec (p-name b-vars p-body saved-env) ; here
                      (if (eqv? search-sym p-name)
                          (proc-val (procedure
                                     b-vars ; here
                                     p-body
                                     env))
                          (apply-env saved-env search-sym))))))
```

Exercise 3.32

```
;; the-grammar
(expression
 ("letrec"
  (arbno identifier "(" identifier ")" "=" expression)
  "in" expression)
 letrec-exp)
```

```
;; value-of
(letrec-exp (p-names b-vars p-bodies letrec-body)
            (value-of letrec-body
                      (extend-env-rec p-names
                                      b-vars
                                      p-bodies
                                      env)))

(define-datatype environment environment?
  (empty-env)
  (extend-env
   (bvar symbol?)
   (bval expval?)
   (saved-env environment?))
  (extend-env-rec
   (ids (list-of symbol?))
   (bvars (list-of symbol?))
   (bodies (list-of expression?))
   (saved-env environment?)))

;; apply-env : Env * Var -> ExpVal
(define apply-env
  (lambda (env search-sym)
    (cases environment env
      (empty-env ()
                 (eopl:error 'apply-env
                             "No binding for ~s"
                             search-sym))
      (extend-env (var val saved-env)
                  (if (eqv? search-sym var)
                      val
                      (apply-env saved-env search-sym)))
      (extend-env-rec (p-names b-vars p-bodies saved-env)
                      (let ((components
                             (rec-search
                              search-sym
                              p-names
                              b-vars
                              p-bodies)))
                        (if components
                            (proc-val
                             (procedure
                              (car components)
                              (cdr components)
                              (extend-env-rec
                               p-names
                               b-vars
                               p-bodies
                               saved-env)))
                            (apply-env saved-env search-sym))))))

;; rec-search :
;; Var * Listof(Var) * Listof(Var) * Listof(Exp) -> Cons(Var, Exp)
(define rec-search
  (lambda (search-sym p-names b-vars p-bodies)
    (cond ((null? p-names) #f)
          ((eqv? search-sym (car p-names))
           (cons (car b-vars) (car p-bodies)))
          (else (rec-search search-sym
                            (cdr p-names)
                            (cdr b-vars)
                            (cdr p-bodies))))))
```

Exercise 3.33

We have a prettier clause for `extend-env-rec` in this exercise. In some places the variable names can be changed to denote a list of lists. I have not done so. Procedures not made from `letrec` are also changed to satisfy the `procedure` constructor, but shall remain unary.

```
;; the-grammar
(expression
 ("(" expression (arbno expression) ")")
 call-exp)

(expression
 ("letrec"
  (arbno identifier "(" (arbno identifier) ")" "=" expression)
  "in" expression)
 letrec-exp)

;; value-of
(proc-exp (var body)
          (proc-val (procedure (list var) body env)))
(call-exp (rator rands)
          (let ((proc (expval->proc (value-of rator env)))
                (args (map (lambda (rand) (value-of rand env))
                           rands)))
            (apply-procedure proc args)))

;; apply-procedure : Proc * Listof(ExpVal) -> ExpVal
(define apply-procedure
  (lambda (proc1 args)
    (cases proc proc1
      (procedure (vars body saved-env)
                 (value-of body (extend-env* vars args saved-env))

;; proc? : SchemeVal -> Bool
;; procedure : Listof(Var) * Exp * Env -> Proc
```

```
(define-datatype proc proc?
  (procedure
   (bvar (list-of symbol?))
   (body expression?)
   (env environment?)))

(define-datatype environment environment?
  (empty-env)
  (extend-env
   (bvar symbol?)
   (bval expval?)
   (saved-env environment?))
  (extend-env-rec
   (ids (list-of symbol?))
   (bvars (list-of (list-of symbol?)))
   (bodies (list-of expression?))
   (saved-env environment?)))

;; apply-env : Env * Var -> ExpVal
(define apply-env
  (lambda (env search-sym)
    (cases environment env
      (empty-env ()
                 (eopl:error ’apply-env
                             "No binding for ~s"
                             search-sym))
      (extend-env (var val saved-env)
                  (if (eqv? search-sym var)
                      val
                      (apply-env saved-env search-sym)))
      (extend-env-rec (p-names b-vars p-bodies saved-env)
                      (let ((p-val
                             (apply-rec-env search-sym
                                            p-names b-vars
                                            p-bodies
                                            env)))
                        (if p-val
                            p-val
                            (apply-env saved-env search-sym)))))))

;; apply-rec-env :
;; Var * Listof(Var) * Listof(Listof(Var)) * Listof(Exp) * Env
;; -> ExpVal
(define apply-rec-env
  (lambda (search-sym p-names b-vars p-bodies enclosing-env)
    (cond ((null? p-names) #f)
          ((eqv? search-sym (car p-names))
           (proc-val (car b-vars) (car p-bodies) enclosing-env))
          (else (apply-rec-env search-sym
                               (cdr p-names)
                               (cdr b-vars)
                               (cdr p-bodies)
                               enclosing-env)))))
```

Exercise 3.34

Rather than leaving `empty-env` and `extend-env` to be part of the data type definition and `extend-env-rec` to be procedural, we convert all three of them to be procedural. Since environments are procedures, then the predicate is defined as such, although procedures, poorly, share the same predicate implementation. An environment is that which takes an Env and Var and returns an ExpVal. They must take an Env because `extend-env-rec` needs a reference to its own environment.

```
;; environment? : SchemeVal -> Bool
(define environment? procedure?)
```

```
;; apply-env : Env * Var -> ExpVal
(define apply-env
  (lambda (env search-var)
    (env env search-var)))

;; empty-env : () -> Env
(define empty-env
  (lambda ()
    (lambda (env search-var)
      (eopl:error ’empty-env "~s not bound" search-var))))

;; extend-env : Var * ExpVal * Env -> Env
(define extend-env
  (lambda (var val saved-env)
    (lambda (env search-var)
      (if (eqv? search-var var)
          val
          (apply-env saved-env search-var)))))

;; extend-env-rec : Var * Var * Exp * Env -> Env
(define extend-env-rec
  (lambda (p-name b-var p-body saved-env)
    (lambda (env search-var)
      (if (eqv? search-var p-name)
          (proc-val (procedure b-var p-body env))
          (apply-env saved-env search-var)))))
```

Exercise 3.35

Since `extend-env-rec` is defined in terms of `extend-env`, we need only dispatch on the type of the value held in `extend-env`.

```
(define-datatype environment environment?
  (empty-env)
  (extend-env
   (bvar symbol?)
   (bval (lambda (x) (or (vector? x) (expval? x))))
   (saved-env environment?)))

;; extend-env-rec : Var * Var * Exp * Env -> Env
(define extend-env-rec
  (lambda (p-name b-var body saved-env)
    (let ((vec (make-vector 1)))
      (let ((new-env (extend-env p-name vec saved-env)))
        (vector-set! vec 0 (proc-val (procedure b-var
                                                body
                                                new-env)))
        new-env))))

;; apply-env : Env * Var -> ExpVal
(define apply-env
  (lambda (env search-sym)
    (cases environment env
      (empty-env ()
                 (eopl:error ’apply-env
                             "No binding for ~s"
                             search-sym))
      (extend-env (var val saved-env)
                  (if (eqv? search-sym var)
                      (if (vector? val)
                          (vector-ref val 0)
                          val)
                      (apply-env saved-env search-sym))))))
```

Exercise 3.36

We modify `extend-env-rec` to use the vector implementation, setting each index in the vector to be a `proc-val`. Now `new-env` takes a list as its first argument, so we change the `extend-env` constructor to take either a bound variable or a list of bound variables, since this constructor is shared with `extend-env-rec`. For `apply-env`, we are either looking at a vector or a symbol. If `val` is a vector, then we find the index of procedure name, if it exists, and reference that index to get the `proc-val`. If `val` is not a vector, then it is a symbol, and we need only compare by equivalence.

```
;; the-grammar
;; mutually recursive unary procedures
(expression
 ("letrec"
  (arbno identifier "(" identifier ")" "=" expression)
  "in" expression)
 letrec-exp)

;; value-of
;; name change
(letrec-exp (p-names b-vars p-bodies letrec-body)
            (value-of letrec-body
                      (extend-env-rec p-names b-vars p-bodies env)

(define-datatype environment environment?
  (empty-env)
  (extend-env
   (bvar (lambda (x) (or (symbol? x) ((list-of symbol?) x))))
   (bval (lambda (x) (or (vector? x) (expval? x))))
   (saved-env environment?)))

;; extend-env-rec :
;; Listof(Var) * Listof(Var) * Listof(Exp) * Env -> Env
(define extend-env-rec
  (lambda (p-names b-vars bodies saved-env)
    (let* ((len (length p-names))
           (vec (make-vector (length p-names)))
           (new-env (extend-env p-names vec saved-env)))
      (for-each (lambda (index b-var body)
                  (vector-set! vec
                               index
                               (proc-val (procedure b-var
                                                    body
                                                    new-env))))
                (enumerate-interval 0 (- len 1))
                b-vars
                bodies)
      new-env)))

;; enumerate-interval : Int * Int -> Listof(Int)
(define enumerate-interval
  (lambda (a b)
    (if (> a b)
        ’()
        (cons a (enumerate-interval (+ a 1) b)))))

;; apply-env : Env * Var -> ExpVal
(define apply-env
  (lambda (env search-sym)
    (cases environment env
      (empty-env ()
                 (eopl:error ’apply-env
                             "No binding for ~s"
                             search-sym))
      (extend-env (var/s val saved-env)
                  (if (vector? val)
                      (let ((index (index-of search-sym var/s)))
                        (if index
                            (vector-ref val index)
                            (apply-env saved-env search-sym)))
                      (if (eqv? search-sym var/s)
                          val
                          (apply-env saved-env search-sym)))))))

;; index-of : Var * Listof(Var) -> Int + Bool
(define index-of
  (lambda (sym vars)
    (define iter
      (lambda (sym vars index)
        (cond ((null? vars) #f)
              ((eqv? sym (car vars)) index)
              (else (iter sym (cdr vars) (+ index 1))))))
    (iter sym vars 0)))
```

Exercise 3.37

```
let even(x) = if zero?(x) then 1 else (odd -(x,1))
in let odd(x) = if zero?(x) then 0 else (even -(x,1))
in (odd 3)
```