Exercise 3.38

```
;; the-grammar
(expression
 ("cond" (arbno expression "==>" expression) "end")
 cond-exp)

;; translation-of
(cond-exp (predicates consequents)
          (cond-exp (map (lambda (p) (translation-of p senv))
                         predicates)
                    (map (lambda (c) (translation-of c senv))
                         consequents)))

;; value-of
(cond-exp (predicates consequents)
          (value-of-cond predicates consequents nameless-env))

;; value-of-cond :
;; Listof(Exp) * Listof(Exp) * Nameless-env -> ExpVal
(define value-of-cond
  (lambda (predicates consequents nameless-env)
    (cond ((null? predicates)
           (eopl:error 'value-of-cond
                       "no true predicates in cond"))
          ((expval->bool (value-of (car predicates) nameless-env))
           (value-of (car consequents) nameless-env))
          (else (value-of-cond (cdr predicates)
                               (cdr consequents)
                               nameless-env)))))
```

Exercise 3.39

This does not check the condition that the number of variables in an unpack expression matches the number of elements in the corresponding

list.

```
;; the-grammar
(expression
 ("cons" "(" expression "," expression ")")
 cons-exp)

(expression
 ("car" "(" expression ")")
 car-exp)

(expression
 ("cdr" "(" expression ")")
 cdr-exp)

(expression
 ("null?" "(" expression ")")
 null?-exp)

(expression
 ("emptylist")
 emptylist-exp)

(expression
 ("list" "(" (separated-list expression ",") ")")
 list-exp)

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

(expression
 ("%unpack" expression "in" expression)
```

```
 nameless-unpack-exp)

;; translation-of
(cons-exp (exp1 exp2)
          (cons-exp (translation-of exp1 senv)
                    (translation-of exp2 senv)))
(car-exp (exp1) (car-exp (translation-of exp1 senv)))
(cdr-exp (exp1) (cdr-exp (translation-of exp1 senv)))
(null?-exp (exp1) (null?-exp (translation-of exp1 senv)))
(emptylist-exp () (emptylist-exp))
(list-exp (exps)
          (list-exp (map (lambda (exp)
                           (translation-of exp senv))
                         exps)))
(unpack-exp (vars exp1 body)
            (nameless-unpack-exp
             (translation-of exp1 senv)
             (translation-of body (extend-senv* vars senv))))

;; value-of
(cons-exp (exp1 exp2)
          (pair-val (value-of exp1 nameless-env)
                    (value-of exp2 nameless-env)))
(car-exp (exp1) (car (expval->pair (value-of exp1 nameless-env))))
(cdr-exp (exp1) (cdr (expval->pair (value-of exp1 nameless-env))))
(null?-exp (exp1)
           (let ((val1 (value-of exp1 nameless-env)))
             (cases expval val1
               (emptylist-val () (bool-val #t))
               (else (bool-val #f)))))
(emptylist-exp () (emptylist-val))
(list-exp (exps) (value-of-list exps nameless-env))
```

```
(nameless-unpack-exp (exp1 body)
                     (value-of
                      body
                      (extend-nameless-env*
                       (expval->list (value-of exp1 nameless-env))
                       nameless-env)))

;; value-of-list : Listof(Nameless-exp) * Nameless-env -> ExpVal
(define value-of-list
  (lambda (exps nameless-env)
    (if (null? exps)
        (emptylist-val)
        (pair-val (value-of (car exps) nameless-env)
                  (value-of-list (cdr exps) nameless-env)))))
```

Exercise 3.40

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

(expression
 ("%lexrec" expression "in" expression)
 nameless-letrec-exp)
(expression
 ("%nameless-letrec-var" number)
 nameless-letrec-var-exp)

;; translation-of
(var-exp (var)
         (if (eqv? 'letrec (var-context senv var))
             (nameless-letrec-var-exp (apply-senv senv var))
```

```
              (nameless-var-exp (apply-senv senv var))))

(letrec-exp (p-name b-var p-body letrec-body)
            (nameless-letrec-exp
             (translation-of
              p-body
              (extend-senv
               b-var
               (extend-senv-letrec p-name senv)))
             (translation-of
              letrec-body
              (extend-senv-letrec p-name senv))))

;; value-of
(nameless-letrec-exp (p-body letrec-body)
                     (value-of
                      letrec-body
                      (extend-nameless-env
                       (proc-val (procedure p-body nameless-env))
                       nameless-env)))

(nameless-letrec-var-exp (n)
                         (let ((nenv (nameless-env-after
                                      nameless-env
                                      n))
                               (proc1 (expval->proc
                                       (apply-nameless-env
                                        nameless-env
                                        n))))
                           (cases proc proc1
                             (procedure (body env)
                                        (proc-val
```

```
                                                    (procedure body
                                                              nenv))))))

;;; static environments

;; empty-senv : () -> Senv
(define empty-senv
  (lambda ()
    '()))

;; extend-senv : Var * Senv -> Senv
(define extend-senv
  (lambda (var senv)
    (cons (cons 'not-letrec var) senv)))

;; extend-senv-letrec : Var * Senv -> Senv
(define extend-senv-letrec
  (lambda (var senv)
    (cons (cons 'letrec var) senv)))

;; apply-senv : Senv * Var -> Lexaddr
(define apply-senv
  (lambda (senv var)
    (cond ((null? senv) (report-unbound-var var))
          ((eqv? var (cdar senv)) 0)
          (else (+ 1 (apply-senv (cdr senv) var))))))

;; var-context : Senv * Var -> Sym
(define var-context
  (lambda (senv var)
    (cond ((null? senv) (report-unbound-var var))
          ((eqv? var (cdar senv)) (caar senv))
```

```
          (else (var-context (cdr senv) var)))))

;;; nameless environment

;; nameless-env-after : Nameless-env * Lexaddr -> Nameless-env
(define nameless-env-after
  (lambda (nameless-env n)
    (cond ((zero? n) nameless-env)
          ((null? nameless-env)
           (eopl:error 'nameless-env-up-to
                       "nameless-env not long enough"))
          (else (nameless-env-after (cdr nameless-env)
                                    (- n 1))))))
```

Exercise 3.41

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

(expression
 ("proc" "(" (arbno identifier) ")" expression)
 proc-exp)

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

(expression ("%nameless-var" number number) nameless-var-exp)

(expression
 ("%let" (arbno expression) "in" expression)
 nameless-let-exp)
```

```
;; translation-of
(var-exp (var)
         (let ((lexaddr (apply-senv senv var)))
           (nameless-var-exp (contours lexaddr)
                             (position lexaddr))))
(let-exp (vars exps body)
         (nameless-let-exp
          (map (lambda (exp) (translation-of exp senv))
               exps)
          (translation-of body
                          (extend-senv* vars senv))))
(proc-exp (vars body)
          (nameless-proc-exp
           (translation-of body
                           (extend-senv* vars senv))))
(call-exp (rator rands)
          (call-exp
           (translation-of rator senv)
           (map (lambda (rand) (translation-of rand senv))
                rands)))

;; value-of
(call-exp (rator rands)
          (let ((proc (expval->proc (value-of rator nameless-env))
                (args (map (lambda (rand)
                             (value-of rand nameless-env))
                           rands)))
            (apply-procedure proc args)))
(nameless-var-exp (contours position)
                  (apply-nameless-env nameless-env
                                      contours
```

```
                                        position))
(nameless-let-exp (exps body)
                  (let ((vals (map (lambda (exp)
                                     (value-of exp nameless-env))
                                   exps)))
                    (value-of
                     body
                     (extend-nameless-env* vals nameless-env))))

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

;;; static environments

;; make-lexaddr : Num * Num -> Lexaddr
(define make-lexaddr cons)

;; contours : Lexaddr -> Num
(define contours car)

;; position : Lexaddr -> Num
(define position cdr)

;; empty-senv : () -> Senv
(define empty-senv
  (lambda ()
```

```
    '()))

;; extend-senv : Var * Senv -> Senv
(define extend-senv
  (lambda (var senv)
    (cons (list var) senv)))

;; extend-senv* : Listof(Var) * Senv -> Senv
(define extend-senv*
  (lambda (vars senv)
    (cons vars senv)))

;; apply-senv : Senv * Var -> Lexaddr
(define apply-senv
  (lambda (senv var)
    (apply-senv-iter senv var 0)))

;; apply-senv-iter : Senv * Var * Num -> Lexaddr
(define apply-senv-iter
  (lambda (senv var contours)
    (if (null? senv)
        (report-unbound-var var)
        (let ((position (apply-senv-contour (car senv) var 0)))
          (if position
              (make-lexaddr contours position)
              (apply-senv-iter (cdr senv)
                               var
                               (+ contours 1)))))))

;; apply-senv-contour : Senv * Var * Num -> Num
(define apply-senv-contour
  (lambda (senv var position)
```

```
    (cond ((null? senv) #f)
          ((eqv? var (car senv)) position)
          (else (apply-senv-contour (cdr senv)
                                    var
                                    (+ position 1))))))

;;; nameless environments

;; nameless-environment? : SchemeVal -> Bool
(define nameless-environment?
  (lambda (x)
    ((list-of (list-of expval?)) x)))

;; empty-nameless-env : () -> Nameless-env
(define empty-nameless-env
  (lambda ()
    '()))

;; empty-nameless-env? : Nameless-env -> Bool
(define empty-nameless-env?
  (lambda (x)
    (null? x)))

;; extend-nameless-env : ExpVal * Nameless-env -> Nameless-env
(define extend-nameless-env
  (lambda (val nameless-env)
    (cons (list val) nameless-env)))

;; extend-nameless-env* :
;; Listof(ExpVal) * Nameless-env -> Nameless-env
(define extend-nameless-env*
  (lambda (vals nameless-env)
```

```
    (cons vals nameless-env)))

;; apply-nameless-env : Nameless-env * Num * Num -> ExpVal
(define apply-nameless-env
  (lambda (nameless-env contours position)
    (cond ((null? nameless-env)
           (eopl:error 'apply-nameless-env
                       "unbound variable"))
          ((zero? contours)
           (list-ref (car nameless-env) position))
          (else (apply-nameless-env (cdr nameless-env)
                                    (- contours 1)
                                    position)))))
```