Scheme Translation of Functions from

FUNCTIONAL PROGRAMMING, APPLICATION AND IMPLEMENTATION

by Dave Laymon

Computer Science Department

Indiana University, Bloomington, Indiana 47405

TECHNICAL REPORT NO. 139

Scheme Translation of Functions from

FUNCTIONAL PROGRAMMING APPLICATION

AND IMPLEMENTATION

by Dave Laymon

Spring, 1983

Scheme Translation of Functions from

FUNCTIONAL PROGRAMMING, APPLICATION AND IMPLEMENTATION

by Peter Henderson

Prentice-Hall, London, 1980

by Dave Laymon

Indiana University, Bloomington

Spring, 1983

INTRODUCTION

With the current trends toward smaller, faster, and less-expensive hardware, it is frequently desirable to build programs which are more easily understood, even if effeciency is somewhat decreased. The use of purely functional languages allows one to write clearer and shorter programs than equivalent ones written in more conventional languages such as Fortran or Pascal.

In FUNCTIONAL PROGRAMMING Henderson has produced a collection of concepts concerning the techniques involved in writing functional programs. The implementation of functional languages is also discussed. The author uses a general, abstract form of Lisp as a vehicle from which he develops these topics.

The purpose of this document is to implement the abstract forms of Henderson's functions in a concrete language: Scheme 311. It is felt that the reader of FUNCTIONAL PROGRAMMING who has some experience with Scheme 311 will derive greater appreciation and understanding of functional programming and of Scheme 311 in particular, with this translation.

To avoid ambiguity, different versions of the same function (or similar functions) have generally been named by concatenating a version number on the end of the name. For instance, different versions of the function "member" are called "member1", "member2", ...

It is not practical to implement "where" and "whererec" in the form used by Henderson;

```
(<body> (where ((<var1> <form1>)
                (<var2> <form2>)
                      .
                      .
                      .
             )))
```

because functions in Scheme are called in the form:

```
(<function name> <argument>)          .
```

Therefore, "let" and "letrec" are used for all local definitions.

```
------------------------------ HELP FUNCTIONS ------------------------------

(define first car)

(define second (lambda (x) (car (cdr x))))

(define third (lambda (x) (car (cdr (cdr x)))))

(define fourth (lambda (x) (car (cdr (cdr (cdr x))))))

(define rest cdr)

(define snoc (lambda (x lis) (append lis (cons x nil))))

------------------------------ SECTION 1.1 ---------------------------------

(define square (lambda (x) (* x x)))

(define reciprocal (lambda (x) (if (0? x) '**error** (/ 1 x))))

(define max1 (lambda (x y) (if (> x y) x y)))

(define largest (lambda (x y z) (max1 (max1 x y) z)))

(define max2
        (lambda (lis)
         (cond ((null? lis) nil)
               ((null? (rest lis)) (first lis))
               ((> (first lis) (max2 (rest lis))) (first lis))
               (t (max2 (rest lis))))))

  ------------------------------- SECTION 2.2 -------------------------------

(define topleft (lambda (s) (first (first s))))

(define topright (lambda (s) (second (first s))))

(define bottomleft (lambda (s) (first (second s))))

(define bottomright (lambda (s) (second (second s))))

(define twolist (lambda (x y) (cons x (cons y nil))))

(define square2 (lambda (a b c d) (twolist (twolist a b) (twolist c d))))

--------------------------------- SECTION 2.3 ------------------------------

(define f (lambda (x) (if (atom? x) nil (first x))))

(define size (lambda (x) (cond ((null? x) 0) ((null? (rest x)) 1) (t 2))))

(define quotrem (lambda (x y) (twolist (/ x y) (mod x y))))

(define distance (lambda (x y) (abs (- x y))))
```

```
---------------------------------- SECTION 2.4 ------------------------------

(define length1 (lambda (x) (if (null? x) 0 (+ 1 (length1 (rest x))))))

(define sum (lambda (x) (if (null? x) 0 (+ (first x) (sum (rest x))))))

(define append1
        (lambda (x y)
         (cond ((null? x) y)
               ((null? y) x)
               (t (cons (first x) (append1 (rest x) y))))))

(define append2
        (lambda (x y) (if (null? x) y (cons (first x) (append2 (rest x) y)))))

-------------------------------- SECTION 2.5 --------------------------------

(define append3 (lambda (x y) (if (null? y) x (appendnonnull? x y))))

(define appendnonnull?
        (lambda (x y)
         (if (null? x) y (cons (first x) (appendnonnull? (rest x) y)))))

(define reverse
        (lambda (x)
         (if (null? x) nil (putatendof (reverse (rest x)) (first x)))))

(define putatendof
        (lambda (x y)
         (if (null? x) (cons y nil) (cons (first x) (putatendof (rest x) y)))))

(define putatendof2 (lambda (x y) (append x (cons y nil))))

(define atom?sin
        (lambda (u)
         (if (null? u) nil (addtoset (atom?sin (rest u)) (first u)))))

(define addtoset (lambda (s x) (if (member x s) s (cons x s))))

(define member1
        (lambda (x s)
         (cond ((null? s) nil)
               ((member x (rest s)) t)
               ((eq? x (first s)) t)
               (t nil))))

(define member2
        (lambda (x s)
         (cond ((null? s) nil)
               ((eq? x (first s)) t)
               ((member2 x (rest s)) t)
               (t nil))))

(define member3
        (lambda (x s)
         (cond ((null? s) nil)
               ((eq? x (first s)) t)
               (t (member3 x (rest s))))))
```

```
(define atomsin2
        (lambda (u)
         (cond ((null? u) nil)
               ((atom? (first u)) (addtoset (atomsin2 (rest u)) (first u)))
               (t (union (atomsin2 (first u)) (atomsin2 (rest u)))))))

(define atomsin3
        (lambda (u)
         (cond ((null? u) nil)
               ((atom? (first u)) (addtoset (atomsin3 (rest u)) (first u)))
               (t (union (atomsin3 (first u)) (atomsin3 (rest u)))))))

(define union
        (lambda (u v)
         (if (null? u) v (addtoset (union (rest u) v) (first u)))))
```

---------------------------------- SECTION 2.6 ------------------------------

```
(define rev (lambda (x y) (if (null? x) y (rev (rest x) (cons (first x) y)))))

(define reverse2 (lambda (x) (rev x nil)))

(define rev2 (lambda (x y) (append (reverse2 x) y)))

(define sumprod (lambda (x) (sp x 0 1)))

(define sp
        (lambda (x s p)
         (if (null? x)
             (twolist s p)
             (sp (rest x) (+ s (first x)) (* p (first x))))))

(define sumprod2
        (lambda (x)
         (if (null? x)
             (twolist 0 1)
             (accumulate (first x) (sumprod2 (rest x))))))

(define accumulate (lambda (n z) (twolist (+ n (first z)) (* n (second z)))))
```

---------------------------------- SECTION 2.7 ------------------------------

```
(define sumprod3
        (lambda (x)
         (if (null? x)
             (twolist 0 1)
             (twolist (+ (first (sumprod3 (rest x))) (first x))
                      (* (first (rest (sumprod3 (rest x)))) (first x))))))

(define sumprod4
        (lambda (x)
         (if (null? x)
             (twolist 0 1)
             (let ((z (sumprod4 (rest x))))
                  (twolist (+ (first z) (first x))
                           (* (second z) (first x)))))))
```

```
(define sumprod5
        (lambda (x)
         (if (null? x)
             (twolist 0 1)
             (let ((n (first x)) (z (sumprod5 (rest x))))
                  (twolist (+ n (first z)) (* n (second z)))))))

(define sumprod6
        (lambda (x)
         (if (null? x)
             (twolist 0 1)
             (let ((n (first x)) (z (sumprod6 (rest x))))
                  (let ((s (first z)) (p (second z)))
                       (twolist (+ n s) (* n p)))))))

(define sum2 (lambda (u v) (cons 'add (cons u (cons v nil)))))

(define prod (lambda (u v) (cons 'mul (cons u (cons v nil)))))

(define diff
        (lambda (e)
         (cond ((atom? e) (if (eq? e 'X) 1 0))
               ((eq? (first e) 'add)
                (let ((e1 (second e)) (e2 (third e)))
                     (sum (diff e1) (diff e2))))
               ((eq? (first e) 'mul)
                (let ((e1 (second e)) (e2 (third e)))
                     (sum (prod e1 (diff e2)) (prod (diff e1) e2))))
               (t '**error**))))
```

----------------------------------- SECTION 2.8 ------------------------------

```
(define inclist
        (lambda (x)
         (if (null? x) nil (cons (+ (first x) 1) (inclist (rest x))))))

(define remlist
        (lambda (x)
         (if (null? x) nil (cons (mod (first x) 2) (remlist (rest x))))))

(define map
        (lambda (x f)
         (if (null? x) nil (cons (f (first x)) (map (rest x) f)))))

(define inc1 (lambda (z) (+ z 1)))

(define inclist2 (lambda (x) (map x inc1)))

(define rem2 (lambda (z) (mod z 2)))

(define remlist2 (lambda (x) (map x rem2)))

(define add (lambda (y z) (+ y z)))

(define reduce
        (lambda (x g a) (if (null? x) a (g (first x) (reduce (rest x) g a)))))
```

```
(define sum3 (lambda (x) (reduce x add 0)))

(define mul (lambda (y z) (* y z)))

(define product1 (lambda (x) (reduce x mul 1)))

(define sumprod7 (lambda (x) (reduce x accumulate (twolist 0 1))))

(define inclist3 (lambda (x) (map x (lambda (z) (+ z 1)))))

(define remlist3 (lambda (x) (map x (lambda (z) (mod z 2)))))

(define sum4 (lambda (x) (reduce x (lambda (y z) (+ y z)) 0)))

(define inclist4 (lambda (x) (let ((inc1 (lambda (z) (+ z 1)))) (map x inc1))))

(define sumprod8
        (lambda (x)
         (let ((sp
                (lambda (x s p)
                 (if (null? x)
                     (twolist s p)
                     (sp (rest x) (+ s (first x)) (* p (first x)))))))
              (sp x 0 1))))

(define atomsin4
        (lambda (lis)
         (letrec ((amsin
                   (lambda (w)
                    (cond ((null? w) nil)
                          ((atom? (first w))
                           (addtoset2 (amsin (rest w)) (first w)))
                          (t (union (amsin (first w)) (amsin (rest w)))))))
                  (union
                   (lambda (u v)
                    (if (null? u)
                        v
                        (addtoset2 (union (rest u) v) (first u)))))
                  (addtoset2 (lambda (s x) (if (member x s) s (cons x s))))
                  (member
                   (lambda (x s)
                    (cond ((null? s) nil)
                          ((eq? x (first s)) t)
                          (t (member x (rest s)))))))
                 (amsin lis))))

(define inc3 (lambda (n) (lambda (z) (+ z n))))

(define inclist5 (lambda (x) (map x (inc3 1))))

(define dot (lambda (f g) (lambda (x) (f (g x)))))

(define increv (lambda (x) (f (g x))))

--------------------------------- SECTION 2.9 ------------------------------

(define atomsize
        (lambda (s)
         (if (atom? s) 1 (+ (atomsize (first s)) (atomsize (rest s))))))
```

```
(define equal1
        (lambda (x y)
         (cond ((atom? x) (eq? x y))
               ((atom? y) (eq? x y))
               ((equal1 (first x) (first y)) (equal1 (rest x) (rest y)))
               (t nil))))

(define equal2
        (lambda (x y)
         (cond ((atom? x) (eq? x y))
               ((atom? y) (eq? x y))
               (t
                (let ((t1 (equal2 (first x) (first y)))
                      (t2 (equal2 (rest x) (rest y))))
                     (if t1 t2 nil))))))
```

--------------------------------- SECTION 3.1 ------------------------------

```
(define assoc1
        (lambda (v a)
         (if (eq? v (first (first a)))
             (second (first a))
             (assoc1 v (rest a)))))

(define assoc2
        (lambda (v a)
         (cond ((null? a) 'undefined)
               ((eq? v (first (first a))) (second (first a)))
               (t (assoc2 v (rest a))))))

(define dim
        (lambda (e a)
         (if (atom? e)
             (assoc2 e a)
             (let ((d1 (dim (second e) a)) (d2 (dim (third e) a)))
                  (cond ((eq? (first e) 'add)
                         (if (same d1 d2) d1 'undefined))
                        ((eq? (first e) 'sub)
                         (if (same d1 d2) d1 'undefined))
                        ((eq? (first e) 'mul) (product2 d1 d2))
                        ((eq? (first e) 'div) (ratio d1 d2))
                        (t 'undefined))))))

(define same
        (lambda (d1 d2)
         (cond ((eq? d1 'undefined) nil)
               ((eq? d2 'undefined) nil)
               ((eq? (mass d1) (mass d2))
                (if (eq? (length d1) (length d2))
                    (eq? (time d1) (time d2))
                    nil))
               (t nil))))

(define mass (lambda (lis) (first lis)))

(define length2 (lambda (lis) (second lis)))

(define time (lambda (lis) (third lis)))
```

```
(define product2
        (lambda (d1 d2)
         (if (or (eq? d1 'undefined) (eq? d2 'undefined))
             'undefined
             (threelist (+ (mass d1) (mass d2))
                        (+ (length2 d1) (length2 d2))
                        (+ (time d1) (time d2))))))

(define ratio
        (lambda (d1 d2)
         (if (or (eq? d1 'undefined) (eq? d2 'undefined))
             'undefined
             (threelist (- (mass d1) (mass d2))
                        (- (length2 d1) (length2 d2))
                        (- (time d1) (time d2))))))

(define threelist (lambda (x y z) (cons x (cons y (cons z nil)))))

(define reduce2
        (lambda (d) (if (eq? d 'undefined) 'undefined (red d))))

(define red
        (lambda (d)
         (if (null? (first d)) d)
         (let ((dprime (red (twolist (rest (num d)) (den d)))))
              (if (member (first (num d)) (den dprime))
                  (twolist (num dprime)
                           (remove (first (num d)) (den dprime)))
                  (twolist (cons (first (num d)) (num dprime))
                           (den dprime))))))

(define num (lambda (d) (first d)))

(define den (lambda (d) (second d)))

(define product3
        (lambda (d1 d2)
         (if (or (eq? d1 'undefined) (eq? d2 'undefined))
             'undefined
             (reduce2
              (twolist (append (num d1) (num d2))
                       (append (den d1) (den d2)))))))

(define ratio2
        (lambda (d1 d2)
         (if (or (eq? d1 'undefined) (eq? d2 'undefined))
             'undefined
             (reduce2
              (twolist (append (num d1) (den d2))
                       (append (den d1) (num d2)))))))

(define same2
        (lambda (d1 d2)
         (if (or (eq? d1 'undefined) (eq? d2 'undefined))
             nil
             (let ((d (ratio d1 d2)))
                  (if (null? (num d)) (null? (den d)) nil)))))
```

---------------------------------- SECTION 3.2 ------------------------------

```
(define p1
        (lambda (s) (equal s (list 'C 'A 'D 'B))))

(define find (lambda (s) (if (p1 s) s (findlist (suc s)))))

(define findlist
        (lambda (s)
         (if (null? s)
             'NONE
             (let ((ss (find (first s))))
                  (if (eq? ss 'NONE) (findlist (rest s)) ss)))))

(define find2 (lambda (s) (findlist2 (cons s nil))))

(define findlist2
        (lambda (s)
         (cond ((null? s) 'NONE)
               ((p1 (first s)) (first s))
               (t (findlist2 (append (suc (first s)) (rest s)))))))

(define p2 (lambda (s y) (equal s y)))

(define ap
        (lambda (op s)
         (let ((orientation (first s)) (transform (second s)))
              (let ((a (first orientation))
                    (b (second orientation))
                    (c (third orientation))
                    (d (fourth orientation)))
                   (cond ((eq? op 'R)
                          (twolist (fourlist c a b d) (cons op transform)))
                         ((eq? op 'F)
                          (twolist (fourlist d c b a) (cons op transform)))
                         (t
                          (twolist (fourlist a c b d)
                                   (cons op transform))))))))

(define fourlist (lambda (w x y z) (cons w (cons x (cons y (cons z nil))))))

(define maxdepth '6)

(define suc
        (lambda (s)
         (if (>= (length (second s)) maxdepth)
             nil
             (threelist (ap 'R s) (ap 'F s) (ap 'C s)))))

(define p3 (lambda (s y) (equal (first s) y)))

(define search1 (lambda (x y maxdepth) (find2 (twolist x nil))))
```

```
(define search2
        (lambda (x y maxdepth)
         (letrec ((searching
                   (lambda (x y maxdepth) (find (twolist (x nil)))))
                  (find (lambda (s) (findlist (cons s nil))))
                  (findlist
                   (lambda (s)
                    (cond ((null? s) 'NONE)
                          ((p (first s)) (first s))
                          (t (findlist (append (suc (first s)) (rest s)))))))
                  (suc
                   (lambda (s)
                    (if (>= (length (second s)) maxdepth)
                        nil
                        (threelist (ap 'R s)
                                   (ap 'F s)
                                   (ap 'C s)))))
                  (p (lambda (s y) (equalto (first s) y)))
                  (ap
                   (lambda (op s)
                    (let ((orientation (first s)) (transform (second s)))
                         (let ((a (first orientation))
                               (b (second orientation))
                               (c (third orientation))
                               (d (fourth orientation)))
                              (cond ((eq? op 'R)
                                     (twolist (fourlist c a b d)
                                              (cons op transform)))
                                    ((eq? op 'F)
                                     (twolist (fourlist d c b a)
                                              (cons op transform)))
                                    (t
                                     (twolist (fourlist a c d b)
                                              (cons op transform))))))))
                  (length (lambda (x)
                           (if (null? x) 0 (+ 1 (length (rest x)))))
                          15)
                  (append
                   (lambda (x y)
                    (if (null? x) y (cons (first x) (append (rest x) y)))))
                 (twolist (lambda (x y) (cons x (cons y nil))))
                 (threelist (lambda (x y z) (cons s (cons y (cons z nil)))))
                 (fourlist
                  (lambda (w x y z) (cons w (cons x (cons y (cons z nil))))))
                 (equalto
                  (lambda (x y)
                   (cond ((atom? x) (eq? x y))
                         ((atom? y) (eq? x y))
                         ((equalto (first x) (first y))
                          (equalto (rest x) (rest y)))
                         (t nil))))
                 (searching x y maxdepth))))
(define search3
        (lambda (x y maxdepth)
         (letrec ((suc
                   (lambda (s)
                    (if (>= (length (second s)) maxdepth)
                        nil
                        (threelist (ap 'R)
                                   (ap 'F)
                                   (ap 'C)))))
                  (p (lambda (s y) (equal (first s) y)))
                  (ap
                   (lambda (op s)
                    (let ((orientation (first s)) (transform (second s)))
                         (let ((a (first orientation))
                               (b (second orientation))
                               (c (third orientation))
                               (d (fourth orientation)))
                              (cond ((eq? op 'R)
                                      (twolist (fourlist c a b d)
                                               (cons op transform)))
                                     ((eq? op 'F)
                                      (twolist (fourlist d c b a)
                                               (cons op transform)))
                                     (t
                                      (twolist (fourlist a c d b)
                                               (cons op transform))))))))))
                 (letrec ((find (lambda (s) (findlist (cons s nil))))
                          (findlist
                           (lambda (s)
                            (cond ((null? s) 'NONE)
                                  ((p (first s)) (first s))
                                  (t
                                   (findlist
                                    (append (suc (first s)) (rest s)))))))))
                         (find (twolist x nil))))))
```

```
(define search4
        (lambda (x y maxdepth)
         (let ((ap
                (lambda (op s)
                 (let ((orientation (first s)) (transform (second s)))
                      (let ((a (first orientation))
                            (b (second orientation))
                            (c (third orientation))
                            (d (fourth orientation)))
                           (cond ((eq? op 'R)
                                  (twolist (fourlist c a b d)
                                           (cons op transform)))
                                 ((eq? op 'F)
                                  (twolist (fourlist d c b a)
                                           (cons op transform)))
                                 (t
                                  (twolist (fourlist a c d b)
                                           (cons op transform))))))))
               (p (lambda (s y) (equal (first s) y)))
               (let suc
                    (lambda (s)
                     (if (>= (length (second s)) maxdepth)
                         nil
                         (threelist (ap 'R s)
                                    (ap 'F s)
                                    (ap 'C s)))))
               (letrec ((find (lambda (s) (findlist (cons s nil))))
                        (findlist
                         (lambda (s)
                          (cond ((null? s) 'NONE)
                                ((p (first s)) (first s))
                                (t
                                 (findlist
                                  (append (suc (first s)) (rest s))))))))
                      (find (twolist x nil))))))

(define search5
        (lambda (x y maxdepth)
         (letrec ((find (lambda (s) (findlist (cons s nil))))
                  (findlist
                   (lambda (s)
                    (cond ((null? s) 'NONE)
                          ((p (first s)) (first s))
                          (t (findlist (suc (first s)) (rest s))))))
                (let ((p (lambda (s y) (equal (first s) y)))
                      (suc
                       (lambda (s)
                        (if (>= (length (second s)) maxdepth)
                            nil
                            (threelist (ap 'R s)
                                       (ap 'F s)
                                       (ap 'C s)))))
                      (lookfor (lambda (s p suc) (find s))))
                     (lookfor (twolist (x nil)) p suc)))))
```

```
---------------------------------- SECTION 3.3 -----------------------------

(define singletons1
        (lambda (s)
         (if (atom? s)
             (cons s nil)
             (union2 (diffrn (singletons1 (first s)) (makeset (rest s)))
                     (diffrn (singletons1 (rest s)) (makeset (first s)))))))

(define makeset
        (lambda (s)
         (if (atom? s)
             (cons s nil)
             (union2 (makeset (first s)) (makeset (rest s))))))

(define union2
        (lambda (x y)
         (if (null? x)
             y
             (let ((z (union2 (rest x) y)))
                  (if (member (first x) z) z (cons (first x) z))))))

(define diffrn
        (lambda (x y)
         (if (null? x)
             nil
             (let ((z (diffrn (rest x) y)))
                  (if (member (first x) y) z (cons (first x) z))))))

(define singletons2 (lambda (s) (rest (sing1 s nil nil))))

(define sing1
        (lambda (s a b)
         (if (member s a) (cons a b))
         (if (member s b) (cons (cons s a) (remove s b)) (cons a (cons s b)))
         (let ((p (sing1 (first s) a b)))
              (sing1 (rest s) (first p) (rest p)))))

(define remove
        (lambda (x y)
         (if (eq? x (first y)) (rest y) (cons (first y) (remove x (rest y))))))

(define singletons3 (lambda (s) (rest (sing3 s (lambda (y) nil) nil))))

(define sing3
        (lambda (s a b)
         (if (atom? s)
             (if (a s)
                 (cons a b)
                 (if (member s b)
                     (cons (lambda (y) (if (eq? y s) t (a y))) (remove s b))
                     (cons a (cons s b)))
                 (let ((p (sing3 (first s) a b)))
                      (sing3 (rest s) (first p) (rest p)))))))
```

```
-------------------------------- CHAPTER 4 ----------------------------------

(define append4
        (lambda (x y)
         (letrec ((apnd
                   (lambda (v w)
                    (if (null? v) w (cons (first v) (apnd (rest v) w))))))
                 (apnd x y))))

------------------------------ SECTION 4.3 ----------------------------------

(define assoc3
        (lambda (x n v)
         (if (member3 x (first n))
             (locate x (first n) (first v))
             (assoc3 x (rest n) (rest v)))))

(define locate
        (lambda (x l m)
         (if (eq? x (first l)) (first m) (locate x (rest l) (rest m)))))

(define evlis
        (lambda (l n v)
         (if (null? l) nil (cons (eval (first l) n v) (evlis (rest l) n v)))))

(define vars
        (lambda (d)
         (if (null? d) nil (cons (first (first d)) (vars (rest d))))))

(define exprs
        (lambda (d)
         (if (null? d) nil (append (rest (first d)) (exprs (rest d))))))
```

```
(define eval2
        (lambda (e n v)
         (cond ((atom? e) (assoc3 e n v))
               ((eq? (first e) 'quote) (second e))
               ((eq? (first e) 'car) (first (eval2 (second e) n v)))
               ((eq? (first e) 'cdr) (rest (eval2 (second e) n v)))
               ((eq? (first e) 'atom?) (atom? (eval2 (second e) n v)))
               ((eq? (first e) 'cons)
                (cons (eval2 (second e) n v) (eval2 (third e) n v)))
               ((eq? (first e) 'eq)
                (eq? (eval2 (second e) n v) (eval2 (third e) n v)))
               ((eq? (first e) 'if)
                (let ((e1 (second e)) (e2 (third e)) (e3 (fourth e)))
                     (eval2 (if (eval2 e1 n v) e2 e3) n v)))
               ((eq? (first e) 'lambda)
                (cons (cons (second e) (third e)) (cons n v)))
               ((eq? (first e) 'let)
                (let ((y (vars (rest (rest e))))
                      (z (evlis (exprs (rest (rest e))) n v)))
                     (eval2 (second e) (cons y n) (cons z v))))
               ((eq? (first e) 'letrec)
                (letrec ((y (vars (rest (rest e))))
                         (vv (cons 'PENDING v))
                         (z (evlis (exprs (rest (rest e))) n v)))
                        (eval2 (second e) (cons y n) (rplaca vv z))))
               (t
                (let ((c (eval2 (first e) n v)) (z (evlis (rest e) n v)))
                     (eval2 (rest (first c))
                            (cons (first (first c)) (second c))
                            (cons z (rest (rest c)))))))))

(define apply2
        (lambda (f x)
         (let ((c (eval2 f nil nil)))
              (eval2 (rest (first c))
                     (cons (first (first c)) (second c))
                     (cons x (rest (rest c)))))))
```

---------------------------- SECTION 5.1 ---------------------------------

```
(define operand1 (lambda (x) (car (cdr x))))

(define operand2 (lambda (x) (car (cdr (cdr x)))))

(define number (lambda (x) (car (cdr x))))

(define name (lambda (x) (car (cdr x))))

(define iftest (lambda (x) (car (cdr x))))

(define then (lambda (x) (car (cdr (cdr x)))))

(define else (lambda (x) (car (cdr (cdr (cdr x))))))

(define wtest (lambda (x) (car (cdr x))))

(define wbody (lambda (x) (car (cdr (cdr x)))))

(define lhs (lambda (x) (car (cdr x))))
```

```
(define rhs (lambda (x) (car (cdr (cdr x)))))

(define simplevalue
        (lambda (e)
         (cond ((isconst? e) (number e))
               ((issum? e)
                (+ (simplevalue (operand1 e)) (simplevalue (operand2 e))))
               (t ('**error**)))))

(define issum? (lambda (e) (eq? (first e) 'add)))

(define isvar? (lambda (e) (eq? (first e) 'var)))

(define isconst? (lambda (e) (eq? (first e) 'quote)))

(define isnull? (lambda (e) (eq? (first e) 'skip)))

(define isassignment? (lambda (s) (eq? (first s) 'assign)))

(define iseq? (lambda (e) (eq? (first e) 'eq)))

(define issequence? (lambda (e) (eq? (first e) 'seq)))

(define isconditional? (lambda (e) (eq? (first e) 'if)))

(define isloop? (lambda (e) (eq? (first e) 'while)))

(define update
        (lambda (n v x y)
         (if (eq? (first n) x)
             (cons y (rest v))
             (cons (first v) (update (rest n) (rest v) x y)))))

(define val
        (lambda (e n v)
         (cond ((isvar? e) (assoc3 (name e) n v))
               ((isconst? e) (number e))
               ((issum? e)
                (let ((x (val (operand1 e) n v)) (y (val (operand2 e) n v)))
                     (+ x y)))
               ((iseq? e)
                (let ((x (val (operand1 e) n v)) (y (val (operand2 e) n v)))
                     (eq? x y)))
               (t '**error**))))

(define effect
        (lambda (s n v)
         (cond ((isnull? s) v)
               ((isassignment? s) (update n v (lhs s) (val (rhs s) n v)))
               ((issequence? s) (effect (second s) n (effect (first s) n v)))
               ((isconditional? s)
                (effect (if (val (iftest s) n v) (then s) (else s)) n v))
               ((isloop? s)
                (letrec ((w
                          (lambda (v)
                           (if (val (wtest s) n v)
                               (w (effect (wbody s) n v))
                               v))))
                        (w v)))
               (t '**error**))))
```

```
------------------------------ SECTION 5.3 ---------------------------------

(define q1 (lambda (x) (if (> x 100) x (q1 (* 2 x)))))

(define w1
        (lambda (y x q r)
         (if (>= r x) (w1 y x (add1 q) (- r x)) (fourlist y x q r))))

(define f1 (lambda (y x r) (g1 y x y)))

(define g1
        (lambda (y x r)
         (cond ((>= r x) (g1 y x (- r x)))
               ((not (null? r)) (f1 x r r))
               (t x))))

----------------------------- SECTION 5.4 ---------------------------------

(define fact1 (lambda (n) (if (0? n) 1 (* n (fact1 (sub1 n))))))

(define fact2 (lambda (n) (f2 n 1)))

(define f2 (lambda (n m) (if (0? n) m (f2 (sub1 n) (* m n)))))

----------------------------- SECTION 6.2 ---------------------------------

(define index (lambda (n s) (if (0? n) (first s) (index (sub1 n) (rest s)))))

(define locate2 (lambda (i e) (index (rest i) (index (first i) e))))

----------------------------- SECTION 6.3 ---------------------------------

(define position
        (lambda (x a) (if (eq? x (first a)) 0 (add1 (position x (rest a))))))

(define location
        (lambda (x n)
         (let ((z (location x (rest n))))
              (if (member x (first n))
                  (cons 0 (position x (first n)))
                  (cons (+ (first z) 1) (rest z))))))

----------------------------- SECTION 6.4 ---------------------------------

(define compile2 (lambda (e) (comp e nil nil)))

(define complis
        (lambda (e n c)
         (if (null? e)
             (cons 'LDC (cons nil c))
             (complis (rest e) n (comp (first e) n (cons 'CONS c))))))

(define badif (lambda (a b c) (if a b c)))
```

```
(define comp
        (lambda (e n c)
         (cond ((atom? e) (cons 'LD (cons (location e n) c)))
               ((eq? (first e) 'QUOTE)
                (cons 'LDC (cons (second e) c)))
               ((eq? (first e) 'ADD)
                (comp (second e) n (cons 'ADD c)))
               ((eq? (first e) 'CAR)
                (comp (second e) (cons 'CAR c)))
               ((eq? (first e) 'CONS)
                (comp (third e) n (comp (second e) n (cons 'CONS c))))
               ((eq? (first e) 'IF)
                (let ((thenpt (comp (third e) n 'JOIN))
                      (elsept (comp (fourth e) n 'JOIN)))
                     (comp (second e)
                           n
                           (cons 'SEL
                                 (cons thenpt (cons elsept c))))))
               ((eq? (first e) 'LAMBDA)
                (let ((body
                       (comp (third e) (cons (rest e) n) ('RTN)))
                     (cons 'LDF (cons body c))))
               ((eq? (first e) 'LET)
                (let ((m (cons (vars (rest (rest e))) n))
                      (args (exprs (rest (rest e)))))
                     (let ((body (comp (second e) m ('RTN))))
                          (complis args n (cons body (cons 'AP c))))))
               ((eq? (first e) 'LETREC)
                (let ((m (cons (vars (rest (rest e))) n))
                      (args (exprs (rest (rest e)))))
                     (let ((body (comp (second e) m ('RTN))))
                          (cons 'DUM
                                (complis args
                                         m
                                         (cons 'LDF
                                               (cons body
                                                     (cons 'RAP
                                                           c))))))))
               (t
                (complis (rest e) n (comp (first e) n (cons 'AP c)))))))
```

-------------------------------- SECTION 7.1 ------------------------------

Henderson's "or" is implemented as "amb", while his "none" is implemented as "vanish". Both "amb" and "vanish" are described in

section 11, Parallel Processing, of the [SCHEME 311] Version 4 REFERENCE MANUAL. Here amb takes two closures as its arguments.

```
(let ((done nil))
     (par2
         (lambda (killself)
             (block (define vanish (freeze (killself nil)))
                    (change! done t)))
         (lambda (ignored)
             (letrec ((loop (freeze (if done nil (thaw loop)))))
                     (loop)))))
```

```
(define amb
        (lambda (th1 th2)
         (letrec ((ans nil) (loop (freeze (if (null? ans) (loop) ans))))
                 (car
                  (par2 (lambda (ignored) (change! ans (list (thaw th1))))
                        (lambda (ignored)
                         (par2 (lambda (ignored)
                                (change! ans (list (thaw th2)))
                               (lambda (ignored) (loop))))))))))

(define insert
        (lambda (x a)
         (if (null? a)
             (cons x nil)
             (amb (freeze (cons x a))
                  (freeze (cons (first a) (insert x (rest a))))))))

(define perm
        (lambda (b) (if (null? b) nil (insert (first b) (perm (rest b))))))

(define choice
        (lambda (n)
         (if (eq? n 1) 1 (amb (freeze n) (freeze (choice (sub1 n)))))
         (if (eq? n 1) 1 (amb (freeze n) (freeze (choice (sub1 n)))))))

(define pair1 (lambda (n) (cons (choice n) (choice n))))

(define pair2 (lambda (n) (let ((x (choice n))) (cons x x))))

(define pair3
        (lambda (n)
         (if (eq? n 1)
             (cons 1 1)
             (let ((p (pair3 (sub1 n))))
                  (amb (freeze p)
                       (freeze
                        (amb (freeze (cons (first p) n))
                             (freeze
                              (amb (freeze (cons n (rest p)))
                                   (freeze (cons n n)))))))))))

(define restrictedperm1 (lambda (a) (let ((b (perm a))) (if (p b) b (vanish)))))

(define p
        (lambda (b)
         (cond ((null? b) t)
               ((null? (rest b)) t)
               ((eq? (first b) (second b)) (p (rest b)))
               (t nil))))

(define restrictedperm2
        (lambda (b)
         (if (null? b)
             nil
             (restrictedinsert (first b) (restrictedperm2 (rest b))))))
```

```
(define restrictedinsert
        (lambda (x a)
         (if (null? a)
             (cons x nil)
             (amb (freeze (build2 x a))
                  (freeze (build2 (first a) (restrictedinsert x (rest a))))))))

(define build2 (lambda (x y) (if (eq? x (first y)) (cons x y) (vanish))))

---------------------------- SECTION 7.3 -----------------------------------

(define attacks
        (lambda (i j place)
         (if (null? place)
             nil
             (let ((iprime (first (first place)))
                    (jprime (rest (first place))))
                  (cond ((eq? i iprime) t)
                        ((eq? j jprime) t)
                        ((eq? (+ i iprime) (+j jprime)) t)
                        ((eq? (- i j) (- iprime jprime)) t)
                        (t (attacks i j (rest place))))))))

(define addqueen
        (lambda (i n place)
         (let ((j (choice2 n)))
              (if (attacks i j place)
                  (vanish)
                  (let ((newplace (cons (cons i j) place)))
                       (addqueen (add1 i) n newplace))))))

(define choice2
        (lambda (n)
         (if (eq? n 1) 1 (amb (freeze (choice2 (sub1 n))) (freeze n)))))

(define queensoln (lambda (n) (addqueen 1 n nil)))

(define tryqueen
        (lambda (i j n place)
         (if (attacks i j place)
             nil
             (let ((newplace (cons (cons i i) place)))
                  (if (eq? i n) newplace (addqueen (add1 i) n newplace))))))

(define anyqueen1
        (lambda (i j n place)
         (let ((newplace (tryqueen i j n place)))
              (cond (newplace newplace)
                    ((eq? j n) nil (anyqueen i (add1 j) n place))))))

(define addqueen2 (lambda (i n place) (anyqueen i 1 n place)))

(define any
        (lambda (j n f)
         (let ((s (f j))) (cond (s s) ((eq? j n) nil (any (add1 j) n f))))))

(define anyqueen2
        (lambda (i j n place) (any j n (lambda (j) (tryqueen i j n place)))))
```

```
----------------------------- SECTION 8.1 ----------------------------------
(define sumints (lambda (n) (sum3 (integersbetween 1 n))))

(define sum3 (lambda (x) (if (null? x) 0 (+ (first x) (sum3 (rest x))))))

(define integersbetween
        (lambda (m n) (if (> m n) nil (cons m (integersbetween (add1 m) n)))))

(define adjust (lambda (a b) (if (< a 0) a (+ a (thaw b)))))

(define adjust2
        (lambda (a b) (if (< (thaw a) 0) (thaw a) (+ (thaw a) (thaw b)))))

(define integersbetween2
        (lambda (m n)
         (if (> m n) nil (cons m (freeze (integersbetween2 (add1 m) n))))))

(define sum4
        (lambda (x) (if (null? x) 0 (+ (first x) (sum4 (thaw (rest x)))))))

(define first1
        (lambda (k x)
         (if (0? k) nil (cons (first x) (first1 (sub1 k) (thaw (rest x)))))))

(define integersfrom1 (lambda (m) (cons m (freeze (integersfrom1 (add1 m))))))

(define firstsums (lambda (k) (first1 k (sums 0 (integersfrom1 1)))))

(define sums
        (lambda (a x)
         (cons (+ a (first x))
               (freeze (sums (+ a (first x)) (thaw (rest x)))))))

------------------------------ SECTION 8.2 ---------------------------------

(define first2
        (lambda (k x)
         (if (0? k) nil (cons (first x) (first2 (sub1 k) (rest x))))))

(define integersfrom2
        (lambda (m) (cons m (lambda nil (integersfrom2 (add1 m))))))

----------------------------- SECTION 8.3 ----------------------------------

(define f1 (lambda (a b) (if (> a 0) a b)))

(define f2 (lambda (a b) (if (> a 0) a (thaw b))))

(define badif2 (lambda (a b c) (if a b c)))

(define f3 (lambda (a b) (if (> a 0) a (+ b b))))

(define f4 (lambda (a b) (if (> a 0) a (+ (thaw b) (thaw b)))))

(define triangle
        (lambda (n) (append4 (integersbetween2 1 n) (triangle (add1 n)))))
```

```
(define append4
        (lambda (x y)
         (if (null? x)
             y
             (cons (first x) (delay (append4 (thaw (rest x)) y))))))

(define first2
        (lambda (k x)
         (if (0? k) nil (cons (first x) (first2 (sub1 k) (rest x))))))

(define integersfrom3 (lambda (m) (cons m (integersfrom3 (add1 m)))))

(define samefringe (lambda (t1 t2) (eqlis (flatten t1) (flatten t2))))

(define flatten
        (lambda (t)
         (if (atom? t)
             (cons t nil)
             (append5 (flatten (first t)) (flatten (rest t))))))

(define append5
        (lambda (x y) (if (null? x) y (cons (first x) (append5 (rest x) y)))))

(define eqlis
        (lambda (x y)
         (cond ((null? x) (null? y))
               ((null? y) nil)
               ((eq? (first x) (first y)) (eqlis (rest x) (rest y)))
               (t nil))))

------------------------------- SECTION 8.4 --------------------------------

(define addone (lambda (x) (cons (add1 (first x)) (addone (rest x)))))

(define addlist
        (lambda (x y)
         (cons (+ (first x) (first y)) (addlist (rest x) (rest y)))))

(define merge
        (lambda (x y)
         (cond ((eq? (first x) (first y)) (merge (rest x) y))
               ((< (first x) (first y)) (cons (first x) (merge (rest x) y)))
               (t (cons (first y) (merge x (rest y)))))))

(define mul2 (lambda (x) (cons (* (first x) 2) (mul2 (rest x)))))

(define filter
        (lambda (p y)
         (if (0? (remainder (first y) p))
             (filter p (rest y))
             (cons (first y) (filter p (rest y))))))

(define f5 (lambda (x) (add1 x)))

(define g1 (lambda (x y) (+ x y)))

(define h1 (lambda (x f) (+ x (f x))))

(define twice1 (lambda (f) (lambda (x) (f (f x)))))
```

```
(define fourtimes1 (lambda (g) (lambda (x) (g (g (g (g x)))))))

(define fourtimes2 (lambda (f) (twice1 (twice1 f))))

(define map3
        (lambda (x f)
         (if (null? x) nil (cons (f (first x)) (map3 (rest x) f)))))

(define map4
        (lambda (f)
         (lambda (x)
          (if (null? x) nil (cons (f (first x)) (map f (rest x)))))))

(define map5
        (lambda (f)
         (letrec ((g
                   (lambda (x)
                    (if (null? x) nil (cons (f (first x)) (g (rest x)))))))
                 g)))

(define aplist
        (lambda (fl)
         (lambda (x)
          (if (null? x) nil (cons ((first fl) x) (aplist (rest fl) x))))))

(define increments
        (cons (lambda (v) (add1 v))
              (cons (lambda (v) (+ v 2))
                    (cons (lambda (v) (+ v 3)) nil))))

(define dot (lambda (f g) (lambda (x) (f (g x)))))

(define B (lambda (f) (lambda (g) (lambda (x) (f (g x))))))

(define twice2 (lambda (f) ((B f) f)))

(define fac (lambda (n) (if (0? n) 1 (* n (fac (sub1 n))))))

(define facc1
        (lambda (n c)
         (if (0? n) (c 1) (facc1 (sub1 n) (lambda (z) (c (* n z)))))))

(define facc2
        (lambda (n c)
         (cond ((< n 0) (list 'NEGATIVE n))
               ((0? n) (c 1))
               (t (facc2 (sub1 n) (lambda (z) (c (* n z))))))))

------------------------------ SECTION 9.2 --------------------------------

(define orp (lambda (p q) (lambda (x) (if (p x) t (q x)))))

(define seq?
        (lambda (p q)
         (lambda (x)
          (cond ((null? x) (and (p nil) (q nil))) ((and (p nil) (q x)) t))
          (t (seq? (lambda (y) (p (cons (first x) y) q) (rest x)))))))

(define vowel
        (lambda (lis) (and (eq? (first lis) 'V) (null? (rest lis)))))
```

```
(define consonant
        (lambda (lis) (and (eq? (first lis) 'C) (null? (rest lis)))))

(define csequence
        (letrec ((cseq? (orp consonant (seq? consonant cseq?))))
                cseq?))
(define vsequence
        (letrec ((vseq? (orp vowel (seq? vowel vseq?))))
                vseq?))

(define syllable
   (letrec ((syll (orp (seq? csequence vsequence)
                       (orp (seq? vsequence csequence)
                            (seq? csequence (seq? vsequence csequence))))))
          syll))

------------------------------- SECTION 9.3 --------------------------------

(define line (lambda (a b) (list a b)))

(define picture (lambda (p q) (list p q)))

(define addvecs
        (lambda (v1 v2)
         (list (+ (first v1) (first v2)) (+ (second v1) (second v2)))))

(define mulvec (lambda (i vec) (list (* i (first vec)) (* i (second vec)))))

(define ladder
        (lambda (n a b)
         (if (eq? n 1)
             (line a b)
             (picture (line (list (+ n (first a)) (+ n (second a)))
                            (list (+ n (first b)) (+ n (second b))))
                      (ladder (sub1 n) a b)))))

(define xproj (lambda (a) (list (first a) 0)))

(define yproj (lambda (a) (list 0 (second a))))

(define box
        (lambda (a b)
         (picture (picture (line a (addvecs (xproj a) (yproj b)))
                           (line a (addvecs (xproj b) (yproj a))))
                  (picture (line b (addvecs (xproj a) (yproj b)))
                           (line b (addvecs (xproj b) (yproj a)))))))

(define both (lambda (p q) (lambda (a b) (picture (p a b) (q a b)))))

(define xdisp
        (lambda (k p)
         (lambda (a b)
          (p (list (list (+ k (first (first a))) (second (first a)))
                   (list (+ k (first (second a))) (second (second a))))
             (list (list (+ k (first (first b))) (second (first b)))
                   (list (+ k (first (second b))) (second (second b))))))))
```

```
(define ydisp
        (lambda (k p)
         (lambda (a b)
          (p (list (list (first (first a)) (+ k (second (first a))))
                   (list (first (second a)) (+ k (second (second a)))))
             (list (list (first (first b)) (+ k (second (first b))))
                   (list (first (second b)) (+ k (second (second b)))))))))

(define ddisp
        (lambda (k p) (lambda (a b) (p (advecs (k k) a) (addvecs (k k) b)))))

(define row
        (lambda (n p) (if (eq? n 1) p (both p (xdisp 1 (row (sub1 n) p))))))

(define col
        (lambda (n p) (if (eq? n 1) p (both p (ydisp 1 (row (sub1 n) p))))))

(define xfrac
        (lambda (k p) (lambda (a b) (xdisp (/ (- (first b) (first a)) k) p))))

(define altcol
        (lambda (n p q)
         (if (eq? n 1) p (both p (ydisp 1 (altcol (sub1 n) q p))))))

(define cascade
        (lambda (n p)
         (if (eq? n 1) p (both p (ydisp 1 (xfrac 2 (cascade (sub1 n) p)))))))

(define xdimin
        (lambda (n s p)
         (if (eq? n 1)
             (s 1 p)
             (both (s n p) (xdimin (sub1 n) s (xdisp s p))))))

----------------------------- SECTION 10.1 --------------------------------

(define add1d (lambda (a b c f) (f (/ (plus a b c) m) (mod (plus a b c) m))))

(define addnd
        (lambda (x y f)
         (if (null? (rest x))
             (add1d (first x) (first y) 0 f)
             (addnd (rest x)
                    (rest y)
                    (lambda (ci s)
                     (add1d (first x)
                            (first y)
                            ci
                            (lambda (co d) (f co (cons d s)))))))))

---------------------------------- CHAPTER 11 ------------------------------

(import 'implode)

(define e nil)

(define p 0)

(define ch nil)
```

```
(define token nil)

(define type nil)

(define inbuffer nil)

(define outbuffer nil)

(define result nil)

(define outbufptr 0)

------------------------------ SECTION 11.1 -------------------------------

(define svalue
        (lambda (lis)
         (if ( 0?  p)
             (first lis)
             (block (change! p (sub1 p)) (svalue (rest lis))))))

(define ivalue
        (lambda (lis)
         (if ( 0?  p)
             (first lis)
             (block (change! p (sub1 p)) (ivalue (rest lis))))))

(define issymbol? (lambda (x) (eq? x 'P)))

(define isnumber? (lambda (x) (eq? x 'Q)))

(define initialize
        (lambda (p)
         (cond ((issymbol? p) (change! (svalue p) nil))
               ((isnumber? p) (change! (ivalue p) 0)))))

--------------------------------- SECTION 11.2 -----------------------------

(define control
        (lambda (fn args result)
         (block (getexp fn)
                (getexplist args)
                (change! result (exec fn args))
                (putexp result))))

------------------------------ SECTION 11.3 ------------------------------

(define scan
        (lambda (token type)
         (block (gettoken token type)
                (if (eq? type 'ENDFILE) (change! token '>) nil))))
```

```
(define getexp
        (lambda (e)
         (cond
          ((eq? token '<)
           (block (scan token type) (getexplist e) (scan token type))
           ((eq? type 'NUMERIC)
            (block (change! e (number e))
                   (change! ivalue (tointeger token))
                   (scan token type))
            (t
             (block (change! e symbol)
                    (change! svalue token)
                    (scan token type))))))))

(define getexplist
        (lambda (e)
         (block (change! e 'cons)
                (getexp (first e))
                (cond ((eq? token '!) (block scan (getexp (rest e))))
                      ((eq? token '>) (change! (rest e) nil))
                      (t (getexplist (rest e)))))))

---------------------------- SECTION 11.4 -------------------------------

(define gettoken
        (lambda (token type)
         (letrec ((getchar
                   (lambda (lis)
                    (if (null? lis) nil (change! ch (first lis)))))
                  (getstring
                   (lambda (l)
                    (cond ((null? l) nil)
                          ((or (isletter? (first lis))
                               (number? (first lis)))
                           (cons (first lis) (getstring (rest lis))))
                          (t nil))))
                  (getnumber
                   (lambda (str)
                    (cond ((null? str) nil)
                          ((number? (first str))
                           (cons (first str) (getnumber (rest str))))
                          (t nil)))))
                 (block (change! token nil)
                        (getchar inbuffer)
                        (cond ((null? inbuffer)
                               (change! type 'ENDFILE))
                              ((or (number? ch) (eq? ch '-))
                               (block (change! type 'NUMERIC)
                                      (change! token (getnumber inbuffer))))
                              ((isletter ch)
                               (block (change! type 'ALPHANUMERIC)
                                      (change! token (getstring inbuffer))))
                              (t
                               (block (change! type 'DELIMITER)
                                      (change! ch (snoc ch token))
                                      (getchar inbuffer)))))))
```

```
(define getstring
        (lambda (lis)
         (cond ((null? lis) nil)
               ((or (isletter? (first lis)) (number? (first lis)))
                (cons (first lis) (getstring (rest lis))))
               (t nil))))

(define getnumber
        (lambda (lis)
         (cond ((null? lis) nil)
               ((number? (first lis))
                (cons (first lis) (getnumber (rest lis))))
               (t nil))))

(define isletter?
        (lambda (ch)
         (member3 ch
                  (quote
                   (a b c d e f g h i j k l m n o p q r s t u v w x y z)))))

------------------------------ SECTION 11.5 -------------------------------

(define putexp
        (lambda (e)
         (cond
          ((issymbol? e)
           (puttoken (svalue (e)))
           ((isnumber? e) (puttoken (tointeger (ivalue e))))
           (block (puttoken '<)
                  (putexp (first p))
                  (change! p e)
                  (change! p (rest p))
                  (if (and (issymbol p) (null? (svalue p)))
                      nil
                      (block (puttoken |.|) (putexp p)))
                  (puttoken '>))))))

------------------------------ SECTION 11.6 -------------------------------

(define puttoken
        (lambda (token)
         (if (0? (length token))
             nil
             (block (putchar (first token)) (puttoken (rest token))))))

(define putchar
        (lambda (c)
         (block (if (= outbufptr 80) (forcelineout outbuffer) nil)
                (change! outbufptr (add1 outbufptr))
                (change! outbuffer (cons c outbuffer)))))

(define forcelineout (lambda (outbuffer) (print outbuffer)))
```

```
----------------------------- SECTION 11.7 ---------------------------------

(define tointeger
        (lambda (lis)
         (letrec ((buidnum
                   (lambda (ls)
                    (if (null? ls)
                        0
                        (+ (first ls) (* 10 (buildnum (rest ls))))))))
                 (buildnum (reverse lis)))))

(define tostring (lambda (lis) (implode lis)))
```