Exercise2.17

```
(define (end x)
  (if (null? (cdr x))
      (car x)
      (end (cdr x))))

(define test-list (list 1 2 3 4 5 6))
(end test-list)
;; 6
```

Exercise2.18

```
(define (reverse x)
  (define (reverse-iter x result)
    (if (null? x)
    result
    (reverse-iter (cdr x) (cons (car x) result))))
  (reverse-iter x (list)))

(define test-list (list 1 2 3 4 5 6))
(reverse test-list)
;; (6,5,4,3,2,1)
```

Exercise2.19

```
(define us-coins (list 50 25 10 5 1))
(define uk-coins (list 100 50 20 10 5 2 1 0.5))
(define (except-first-denomination x)
  (cdr x))
(define (no-more? x)
  (null? x))
(define (first-denomination x)
  (car x))
(define
  (cc amount coin-values)
  (cond ((= amount 0) 1)
    ((or (< amount 0)
         (no-more? coin-values))
     0)
    (else
     (+
      (cc
       amount
```

```
        (except-first-denomination
         coin-values))
       (cc
        (-
         amount
         (first-denomination
          coin-values))
        coin-values)))))
(cc 100 us-coins)
```

Does the order of the list coinvalues affect the answer produced by cc? Why or why not? No, because when the order changed, for example it changed to (1, 50, 25, 10, 5), and amount is 100, then in the first recursive step the result is (cc 100 (50, 25, 10, 5)) + (cc 99 (1, 50, 25, 10, 5)), all cases are still included.

Exercise2.20

```
(define (same-parity . x)
  (define (same-parity-iter first x)
    (cond ((null? x)
       '())
      ((= (remainder first 2) (remainder (car x) 2))
       (cons (car x) (same-parity-iter first (cdr x))))
      (else (same-parity-iter first (cdr x)))))
  (same-parity-iter (car x) x))
(same-parity 1 2 3 4 5 6 7)
```

syntax sugar

```
(define (same-parity . x)
  (let parity ((first (car x)) (x x))
       (cond ((null? x) '())
             ((= (remainder (car x) 2) (remainder first 2))
              (cons (car x)
                    (parity first (cdr x))))
             (else (parity first (cdr x))))))
(same-parity 2 1 1 2 3 4 5 6 7)
```

Exercise2.21

map

```
(define (square-list x)
  (map (lambda (x) (* x x))
       x))
```

```
(square-list (list 1 2 3 4))
```

recursive

```
(define (square x)
  (* x x))
(define (square-list items)
  (if (null? items)
      ;; nil not working, use '() instead
      '()
      (cons (square (car items)) (square-list (cdr items)))))

(square-list (list 1 2 3 4))
```

Exercise2.22

the first one (cons 1 '()) (cons 4 (list 1)) (cons 9 (list 4 1)) …

the second (cons null 1) (cons (null 1) 4) (cons ((null 1) 4) 9) …

```
(define
  (square-list items)
  (define
    (iter things answer)
    (if (null? things)
    answer
    (iter
     (cdr things)
     (append answer
           (list (square (car things)))))))
  (iter items '()))
(square-list (list 1 2 3 4))
```

Exercise2.23

```
(define (for-each p x)
  (if (null? x)
      #t
      (and (p (car x)) (for-each p (cdr x)))))
(for-each (lambda (x) (newline) (display x)) (list 57 321 88))
```

Exercise2.24

result printed by the interpreter: (1 (2 (3 4)))

structure: (1 (2 (3 4)))—>(2 (3 4))—>(3 4)—>(4 null)

---

v v v v 1 2 3 4

the list structure viewed as a tree: (1 (2 (3 4))) / \ 1 (2 (3 4)) / \ 2 (3 4) / \ 3 4

Exercise2.25

```
(car (cdr (car (cdr (cdr (list 1 3 (list 5 7) 9))))))
(car (car (list (list 7))))
(car (cadr (cadr (cadr (cadr (cadr (cadr (list 1 (list 2 (list 3 (list
   4 (list 5 (list 6 (list 7)))))))))))))))
```

Exercise2.26

```
(define x (list 1 2 3))
(define y (list 4 5 6))
(append x y)
;; (1 2 3 4 5 6)
(cons x y)
;; ((1 2 3) 4 5 6)
(list x y)
;; ((1 2 3) (4 5 6))
```

Exercise2.27

```
(define (reverse x)
  (if (null? x)
      '()
      (append (reverse (cdr x)) (list (car x)))))
(define x (list (list 1 2) (list 3 4)))
(reverse x)

```

```
(define (deep-reverse x)
  (cond ((null? x) '())
      ((not (pair? x)) x)
      (else
       (append (deep-reverse (cdr x)) (list (deep-reverse (car x)))))))
(deep-reverse x)
(define x (list (list 1 2 (list 3 4)) (list 5 6)))
x
;; ((1 2 (3 4)) (5 6))
(deep-reverse x)
;; ((6 5) ((4 3) 2 1))
```

Exercise2.28

```
(define (fringe x)
  (cond ((null? x) '())
    ((not (pair? x)) (list x))
    (else
     (append (fringe (car x)) (fringe (cdr x))))))
(define x (list (list 1 2) (list 3 4)))
(fringe x)
(fringe (list x x))
```

Exercise2.29

```
(define (make-mobile left right)
  (list left right))
(define (make-branch length structure)
  (list length structure))
(define (left-branch mobile)
  (car mobile))
(define (right-branch mobile)
  (car (cdr mobile)))
(define (branch-length branch)
  (car branch))
(define (branch-structure branch)
  (car (cdr branch)))

;; d.
(define (make-mobile left right)
  (cons left right))
(define (make-branch length structure)
  (cons length structure))
(define (right-branch mobile)
  (cdr mobile))
```

```
(define (branch-structure branch)
  (cdr branch))
(define
  (total-weight mobile)
  (cond ((null? mobile) 0)
    ((not (pair? mobile)) mobile)
    (else
     (+
      (total-weight
       (branch-structure
        (left-branch mobile)))
      (total-weight
       (branch-structure
        (right-branch mobile)))))))
(define
  (balanced mobile)
  (cond ((null? mobile) true)
    ((not (pair? mobile)) true)
    ((= (*
         (branch-length
          (left-branch mobile))
         (total-weight
          (branch-structure
           (left-branch mobile))))
        (*
         (branch-length
          (right-branch mobile))
         (total-weight
          (branch-structure
           (right-branch mobile)))))
     (and (balanced
           (branch-structure
        (left-branch mobile)))
          (balanced
           (branch-structure
        (right-branch mobile)))))
    (else false)))
(define
  m1
  (make-mobile
   (make-branch 4 6)
   (make-branch
    5
    (make-mobile
     (make-branch 3 7)
     (make-branch 9 8)))))
(total-weight m1)
;; 21
(balanced m1)
;; #f
(define
```

```
  m2
  (make-mobile
   (make-branch 1 5)
   (make-branch 5 1)))
(balanced m2)
;; #t
```

After the representation of mobiles changed, we only need to rewrite the *right-branch* and *branch-structure* selector

Exercise2.30

```
(define (square-tree tree)
  (map (lambda (sub-tree)
     (if (pair? sub-tree)
         (square-tree sub-tree)
         (* sub-tree sub-tree)))
       tree))
(square-tree (list 1 (list 2 (list 3 4) 5) (list 6 7)))
;; (1 (4 (9 16) 25) (36 49))
(define (square-tree tree)
  (cond ((null? tree) '())
    ((not (pair? tree)) (* tree tree))
    (else (cons (square-tree (car tree))
            (square-tree (cdr tree))))))
(square-tree (list 1 (list 2 (list 3 4) 5) (list 6 7)))
```

Exercise2.31

```
(define (tree-map procedure tree)
  (cond ((null? tree) '())
    ((not (pair? tree)) (procedure tree))
    (else (cons (tree-map procedure (car tree))
            (tree-map procedure (cdr tree))))))
(define (square x)
  (* x x))
(define (square-tree tree) (tree-map square tree))
(square-tree (list 1 (list 2 (list 3 4) 5) (list 6 7)))
;; (1 (4 (9 16) 25) (36 49))
```

Exercise2.32

```
(define
  (subsets s)
  (if (null? s)
      (list '())
      (let ((rest (subsets (cdr s))))
    (append rest (map (lambda (x)
                 (cons (car s) x)) rest)))))
(define set (list 1 2 3))
(subsets (list 2))
```

```
(null? '())
;; #t
(cons 2 '())
;; (2)
(null? (cdr (list 1)))
```

Exercise2.33

```
(define
  (accumulate
   op
   initial
   sequence)
  (if (null? sequence)
      initial
      (op
       (car sequence)
       (accumulate
    op
    initial
    (cdr sequence)))))
;; abstract border, don't need null? or pair?
(define (map p sequence)
  (accumulate (lambda (x y)
            (cons (p x) y)) '() sequence))
(define (append seq1 seq2)
  ;; nil at the end, add (car seq1) to seq2
  (accumulate cons seq2 seq1))
(append (list 1 2) (list 2 3 (list 2 3)))
;; (1 2 2 3)
(define (length sequence)
  (accumulate (lambda (x y)
        (+ 1 y)) 0 sequence))
(length (list 1 (list 1 2) (list 2)))
;; 3
```

Exercise2.34

```
(define (accumulate op initial sequence) (if (null? sequence) initial (
    op (car sequence) (accumulate op initial (cdr sequence)))))
(define (horner-eval x coefficient-sequence)
  (accumulate (lambda (this-coeff higher-terms)
        (+ this-coeff (* higher-terms x))) 0 coefficient-sequence))
(horner-eval 2 (list 1 3 0 5 0 1))
```

Exercise2.35

```
(define (enumerate-tree tree) (cond ((null? tree) '()) ((not (pair?
    tree)) (list tree)) (else (append (enumerate-tree (car tree)) (
    enumerate-tree (cdr tree))))))
(define (accumulate op initial sequence) (if (null? sequence) initial (
    op (car sequence) (accumulate op initial (cdr sequence)))))
(define (count-leaves t)
  (accumulate + 0 (map (lambda (x) 1) (enumerate-tree t))))
(count-leaves (list 1 (list 2 (list 3 4)) 5))
;; 5
```

Exercise2.36 && 2.37

```
(define (accumulate op initial sequence) (if (null? sequence) initial (
    op (car sequence) (accumulate op initial (cdr sequence)))))
(define (carn sequence)
  (cond ((null? sequence) '())
    (else (append (list (car (car sequence))) (carn (cdr sequence))))))
(define (cdrn sequence)
  (cond ((null? sequence) '())
    (else (cons (cdr (car sequence)) (cdrn (cdr sequence))))))
(define
  (accumulate-n op init seqs)
  (if (null? (car seqs))
      '()
      (cons (accumulate op init (carn seqs))
        (accumulate-n op init (cdrn seqs)))))
(accumulate-n + 0 (list (list 1 2 3) (list 4 5 6) (list 7 8 9) (list 10
    11 12)))
;; (22, 26, 30)

;; matrix operations
(define m1 (list (list 1 2 3) (list 4 5 6) (list 7 8 9)))
(define v1 (list 1 2 3))
```

```
20
21 (define (dot-product v w)
22   (accumulate + 0 (map * v w)))
23 (dot-product (list 1 2 3) (list 1 2 3))
24 ;; 14
25
26 (define (matrix-*-vector m v)
27   (map (lambda (x) (dot-product v x)) m))
28 (matrix-*-vector m1 v1)
29 ;; (14, 32, 50)
30
31 (define (transpose mat)
32   (accumulate-n cons '() mat))
33 (transpose m1)
34 ;; ((1 4 7) (2 5 8) (3 6 9))
35
36 (define (matrix-*-matrix m n)
37   (let ((cols (transpose n)))
38     (map (lambda (x) (matrix-*-vector cols x)) m)))
39 (matrix-*-matrix m1 m1)
```

Exercise2.38 && 2.39

```
1  (define (fold-left op initial sequence)
2    (define (iter result rest)
3      (if (null? rest)
4      result
5      (iter (op result (car rest))
6            (cdr rest))))
7    (iter initial sequence))
8  (define (fold-right op initial sequence) (if (null? sequence) initial (
       op (car sequence) (fold-right op initial (cdr sequence)))))
9
10 ;; (fold-right / 1 (list 1 2 3))
11 ;; 3/2
12 (fold-left / 1 (list 1 2 3))
13 ;; 1/6
14 ;; (fold-right list '() (list 1 2 3))
15 ;; (1 (2 (3 ())))
16 (fold-left list '() (list 1 2 3))
17 ;; ((() 1) 2) 3)
18
19 ;; reverse
20 (define (reverse sequence)
21   (fold-left (lambda (x y)
22            (cons y x)) '() sequence))
23 (reverse (list 1 2 3 4))
24
```

```
(define (reverse sequence)
  (fold-right (lambda (x y)
        (append y (list x))) '() sequence))
(reverse (list 1 2 3 4))
;; (4 3 2 1)
```

fold-right and fold-left will get the same values if (op x y)==(op y x)

Exercise2.40 && 2.41 && 2.42

```
(define (accumulate op initial sequence) (if (null? sequence) initial (op (car sequence) (accumulate op initial (cdr sequence)))))
(define (accumulate-and initial sequence) (if (null? sequence) initial (and (car sequence) (accumulate-and initial (cdr sequence)))))
(define (flatmap proc seq) (accumulate append '() (map proc seq)))
(define (make-pair-sum pair)
  (list (car pair) (cadr pair) (+ (car pair) (cadr pair))))
(define (prime-sum? pair) (prime? (+ (car pair) (cadr pair))))
(define (fermat-test n) (define (try-it a) (= (expmod a n n) a)) (try-it (+ 1 (random (- n 1)))))
(define (expmod base exp m) (cond ((= exp 0) 1) ((even? exp) (remainder (square (expmod base (/ exp 2) m)) m)) (else (remainder (* base (expmod base (- exp 1) m)) m))))
(define (fast-prime? n times) (cond ((= times 0) true) ((fermat-test n) (fast-prime? n (- times 1))) (else false)))
(define (prime? x)
  (fast-prime? x 2))
(define (enumerate-interval low high)
  (if (> low high)
      '()
      (cons low (enumerate-interval (+ low 1) high))))

;; unique-pair
(define (unique-pairs n)
  (flatmap
   (lambda (i)
     (map (lambda (j) (list i j))
      (enumerate-interval 1 (- i 1))))
   (enumerate-interval 1 n)))
(unique-pairs 3)
;; ((2 1) (3 1) (3 2))

(define
  (prime-sum-pairs n)
  (map
   make-pair-sum
   (filter
    prime-sum?
```

```
    (unique-pairs n))))
(prime-sum-pairs 3)

(define (unique-triples n)
  (flatmap
   (lambda (i)
     (flatmap (lambda (j)
        (map (lambda (k) (list i j k))
         (enumerate-interval 1 n)))
      (enumerate-interval 1 n)))
   (enumerate-interval 1 n)))
(unique-triples 2)
;; ((1 1 1) (1 1 2) (1 2 1) (1 2 2) (2 1 1) (2 1 2) (2 2 1) (2 2 2))

;; 8 queens
(define (queens board-size)
  (define (queen-cols k)
    (if (= k 0)
    (list empty-board)
    (filter
     (lambda (positions)
       (safe? k positions))
     (flatmap (lambda (rest-of-queens)
            (map (lambda (new-row)
               (adjoin-position new-row k rest-of-queens))
             (enumerate-interval 1 board-size)))
          (queen-cols (- k 1))))))
  (queen-cols board-size))
(define empty-board (list))
(define (safe? k positions)
  (let ((last (car positions)))
    (accumulate-and true (map (lambda (x) (not (= last x))) (cdr
       positions)))))
(define (adjoin-position new-row k rest-of-queens)
  (append (list new-row) rest-of-queens))
;; (safe? 8 (list 1 2 3 4 5 6 7 7))
(define positions (list 1 1 3 4))
(queens 3)
;; ((3 2 1) (2 3 1) (3 1 2) (1 3 2) (2 1 3) (1 2 3))
```

Exercise2.43

```
(flatmap
 (lambda (new-row)
   (map
    (lambda (rest-of-queens)
      (adjoin-position
       new-row
```

```
        k
        rest-of-queens))
     (queen-cols (- k 1))))
  (enumerate-interval
   1
   board-size))
```

Because flatmap use append, (append list1 list2) will car list1 until list1 is null, and Louis's method have a longer list1, so it will cost n times T

Exercise2.44

```
(define (up-split painter n)
  (if (= n 0)
      painter
      (let ((smaller (up-split painter (- n 1))))
    (below painter (beside smaller smaller)))))
```

Exercise2.45

```
(define right-split (split beside below))
(define (split proc1 proc2)
  (lambda (painter n)
    (if (= n 0)
    painter
    (let ((smaller (split painter (- n 1))))
      (proc1 painter (proc2 smaller smaller))))))
```

Exercise2.46&&2.47&&2.48&&2.49&&2.50&&2.51&&2.52

```
(define (make-vector x y)
  (cons x y))
(define (xcor-vect vector)
  (car vector))
(define (ycor-vect vector)
  (cdr vector))
(define (add-vect vector1 vector2)
  (make-vector (+ (xcor-vect vector1) (xcor-vect vector2))
           (+ (ycor-vect vector1) (ycor-vect vector2))))
(define (sub-vect vector1 vector2)
  (make-vector (- (xcor-vect vector1) (xcor-vect vector2))
           (- (ycor-vect vector1) (ycor-vect vector2))))
```

```
(define (scale-vect scalar vector)
  (make-vector (* scalar (xcor-vect vector))
               (* scalar (ycor-vect vector))))
(define vect1 (make-vector 1 2))
(define vect2 (make-vector 3 4))
(define vect3 (make-vector 5 6))

;; constructor and selector impl 1
(define (make-frame origin edge1 edge2)
  (list origin edge1 edge2))
(define (origin-frame frame)
  (car frame))
(define (edge1-frame frame)
  (car (cdr frame)))
(define (edge2-frame frame)
  (car (cdr (cdr frame))))

;; constructor and selector impl 2
(define (make-frame origin edge1 edge2)
  (cons origin (cons edge1 edge2)))
(define (origin-frame frame)
  (car frame))
(define (edge1-frame frame)
  (car (cdr frame)))
(define (edge2-frame frame)
  (cdr (cdr frame)))

(define frame1 (make-frame vect1 vect2 vect3))
(origin-frame frame1)
(edge1-frame frame1)
(edge2-frame frame1)

;; Exercise2.48
(define (make-segment v1 v2)
  (cons v1 v2))
(define (start-segment segment)
  (car segment))
(define (end-segment segment)
  (cdr segment))
;; (define seg1 (make-segment vect1 vect2))
;; (start-segment seg1)
;; (end-segment seg1)


;; Exercise2.49
(define (for-each p x)
  (if (null? x)
      #t
      (and (p (car x)) (for-each p (cdr x)))))
(define
  (frame-coord-map frame)
```

```
  (lambda (v)
    (add-vect
     (origin-frame frame)
     (add-vect
      (scale-vect
       (xcor-vect v)
       (edge1-frame frame))
      (scale-vect
       (ycor-vect v)
       (edge2-frame frame))))))
(define
  (segments->painter
   segment-list)
  (lambda (frame)
    (for-each
     (lambda (segment)
       (draw-line
    ((frame-coord-map frame)
     (start-segment segment))
    ((frame-coord-map frame)
     (end-segment segment))))
     segment-list)))
(define outline
  (let ((segment-list (list
                (make-segment
            (make-vect 0 0)
            (make-vect 1 0))
                (make-segment
            (make-vect 0 0)
            (make-vect 0 1))
                (make-segment
            (make-vect 1 0)
            (make-vect 1 1))
                (make-segment
            (make-vect 1 1)
            (make-vect 0 1)))))
    (segments->painter segment-list)))

(define x-painter
  (let ((segment-list (list
                (make-segment
            (make-vect 0 0)
            (make-vect 1 1))
                (make-segment
            (make-vect 0 1)
            (make-vect 1 0)))))
    (segments->painter segment-list)))

(define diamond
  (let ((segment-list (list
                (make-segment
```

```
115               (make-vect 0.5 0)
116               (make-vect 1 0.5))
117                  (make-segment
118               (make-vect 0.5 0)
119               (make-vect 0 0.5))
120                  (make-segment
121               (make-vect 0.5 1)
122               (make-vect 1 0.5))
123                  (make-segment
124               (make-vect 0.5 1)
125               (make-vect 0 0.5)))))
126      (segments->painter segment-list)))
127
128  (define wave
129    (let ((segment-list (list
130                  (make-segment
131               (make-vect 0.1 0)
132               (make-vect 0.25 0.6))
133                  (make-segment
134               (make-vect 0.3 0)
135               (make-vect 0.5 0.3))
136                  (make-segment
137               (make-vect 0.7 0)
138               (make-vect 0.5 0.3))
139                  (make-segment
140               (make-vect 0.9 0)
141               (make-vect 0.75 0.6))
142                  (make-segment
143               (make-vect 0.25 0.6)
144               (make-vect 0 0.4))
145                  (make-segment
146               (make-vect 0.75 0.6)
147               (make-vect 0.9 0.4))
148                  (make-segment
149               (make-vect 0.9 0.4)
150               (make-vect 1 0.6))
151                  (make-segment
152               (make-vect 0.6 1)
153               (make-vect 0.7 0.9))
154                  (make-segment
155               (make-vect 0.7 0.9)
156               (make-vect 0.6 0.8))
157                  (make-segment
158               (make-vect 0.6 0.8)
159               (make-vect 0.8 0.7))
160                  (make-segment
161               (make-vect 0.8 0.7)
162               (make-vect 1 0.8))
163                  (make-segment
164               (make-vect 0.4 1)
165               (make-vect 0.3 0.9))
```

```
166                (make-segment
167             (make-vect 0.3 0.9)
168             (make-vect 0.4 0.8))
169                (make-segment
170             (make-vect 0.4 0.8)
171             (make-vect 0 0.6))
172                ;; add smile here
173                (make-segment
174             (make-vect 0.4 0.9)
175             (make-vect 0.5 0.8))
176                (make-segment
177             (make-vect 0.5 0.8)
178             (make-vect 0.6 0.9)))))
179     (segments->painter segment-list)))
180
181 ;; Exercise2.50
182 (define (transform-painter painter origin corner1 corner2)
183   (lambda (frame)
184     (let ((m (frame-coord-map frame)))
185       (let ((new-origin (m origin)))
186     (painter (make-frame
187           new-origin
188           (sub-vect (m corner1) new-origin)
189           (sub-vect (m corner2) new-origin)))))))
190
191 (define (flip-horiz painter)
192   (transform-painter painter
193              (make-vect 1.0 0.0)
194              (make-vect 0.0 0.0)
195              (make-vect 1.0 1.0)))
196
197 (define (rotate90 painter) (transform-painter painter (make-vect 1.0
       0.0) (make-vect 1.0 1.0) (make-vect 0.0 0.0)))
198
199 (define (rotate180 painter)
200   (transform-painter painter
201              (make-vect 1.0 1.0)
202              (make-vect 0.0 1.0)
203              (make-vect 1.0 0.0)))
204
205 (define (rotate270 painter)
206   (transform-painter painter
207              (make-vect 0.0 1.0)
208              (make-vect 0.0 0.0)
209              (make-vect 1.0 1.0)))
210
211 (define
212   (beside painter1 painter2)
213   (let ((split-point (make-vect 0.5 0.0)))
214     (let ((paint-left (transform-painter
215                painter1
```

```
                (make-vect 0.0 0.0)
                split-point
                (make-vect 0.0 1.0)))
      (paint-right (transform-painter
            painter2
            split-point
            (make-vect 1.0 0.0)
            (make-vect 0.5 1.0))))
      (lambda (frame)
    (paint-left frame)
    (paint-right frame)))))

(define
  (below painter1 painter2)
  (let ((split-point (make-vect 0.0 0.5)))
    (let ((paint-below (transform-painter
                painter1
                (make-vect 0.0 0.0)
                (make-vect 1.0 0.0)
                split-point))
      (paint-up (transform-painter
            painter2
            split-point
            (make-vect 1.0 0.5)
            (make-vect 0.0 1.0))))
      (lambda (frame)
    (paint-below frame)
    (paint-up frame)))))

(define
  (below painter1 painter2)
  (rotate270 (beside (rotate90 painter2) (rotate90 painter1))))

;; change pattern constructed by corner split
(define
  (corner-split painter n)
  (if (= n 0)
      painter
      (let ((up (up-split painter (- n 1)))
        (right (right-split painter (- n 1))))
    (let ((top-left up)
          (bottom-right right)
          (corner (corner-split painter (- n 1))))
      (beside
       (below painter top-left)
       (below bottom-right corner))))))

;; make the big Mr. Rogers look outward from each corner of the square
(define (square-limit painter n) (let ((combine4
                    (square-of-four
                     (flip-horiz flip-horiz)
```

```
267                          (flip-horiz identity)
268                          (flip-horiz rotate180)
269                          (flip-horiz flip-vert))))))
```