

Exercises

Exercise 1.15

duple : $Int \times SchemeVal \rightarrow Listof(SchemeVal)$

usage: (duple n x) returns a list having x , n times.

```
(define duple
  (lambda (n x)
    (if (zero? n)
        '()
        (cons x (duple (- n 1) x))))))
```

Exercise 1.16

invert : $Listof(List(SchemeVal, SchemeVal)) \rightarrow Listof(List(SchemeVal, SchemeVal))$

usage: returns a list whose elements e_i are those of lst , but all e_i are reversed.

```
(define invert
  (lambda (lst)
    (if (null? lst)
        '()
        (cons (list (cadar lst) (caar lst))
              (invert (cdr lst))))))
```

Exercise 1.17 **down** : $Listof(SchemeVal) \rightarrow Listof(List(SchemeVal))$

usage: returns a list of the elements of lst , but with each element wrapped in parentheses.

```
(define down
  (lambda (lst)
    (if (null? lst)
        '()
        (cons (list (car lst))
              (down (cdr lst))))))
```

Exercise 1.18

swapper : $Symbol \times Symbol \times S - list \rightarrow S - list$

usage: returns a list having all of the elements in *slist* but with all occurrences of *s1* and *s2* swapped for each other.

```
(define swapper
  (lambda (s1 s2 slist)
    (if (null? slist)
        '()
        (cons (swap-in-s-exp s1 s2 (car slist))
                (swapper s1 s2 (cdr slist))))))
```

swap-in-s-exp : $Symbol \times Symbol \times S - exp \rightarrow S - exp$

usage: returns an s-exp that is a symbol where it is *s2* if *sexp* is *s1*, or *s1* if *sexp* is *s2*, or *sexp* itself if neither, or an s-list otherwise.

```
(define swap-in-s-exp
  (lambda (s1 s2 sexp)
    (if (symbol? sexp)
        (cond ((eqv? sexp s1) s2)
              ((eqv? sexp s2) s1)
              (else sexp))
        (swapper s1 s2 sexp))))
```

Exercise 1.19

list-set : $List \times Int \times SchemeVal \rightarrow List$

usage: returns a list like *lst*, except that the *n*-th element, using zero-based indexing, is *x*

```
(define report-list-too-short
  (lambda (n proc-name)
    (eopl:error proc-name
      "List too short by ~s elements.~%"
      (+ n 1))))

(define list-set
  (lambda (lst n x)
```

```

(cond ((null? lst) (report-list-too-short n 'list-set))
      ((zero? n) (cons x (cdr lst)))
      (else (cons (car lst)
                   (list-set (cdr lst) (- n 1) x))))))

```

Exercise 1.20

count-occurrences : $Symbol \times S - list \rightarrow Int$

usage: (count-occurrences *s* *slist*) = the number of occurrences of *s* in *slist*.

```

(define count-occurrences
  (lambda (s slist)
    (if (null? slist)
        0
        (+ (count-occurrences-in-s-exp s (car slist))
            (count-occurrences s (cdr slist))))))

```

count-occurrences-in-s-exp : $Symbol \times S - exp \rightarrow Int$

usage: if *sexp* is a symbol, then 1 if *sexp* is *s*, 0 otherwise, or the number of occurrences of *s* in *sexp* if *sexp* is a s-list.

```

(define count-occurrences-in-s-exp
  (lambda (s sexp)
    (if (symbol? sexp)
        (if (eqv? sexp s) 1 0)
        (count-occurrences s sexp))))

```

Exercise 1.21 **product** : $Listof(Symbol) \times Listof(Symbol)$

```

(define product
  (lambda (sos1 sos2)
    (if (null? sos1)
        '()
        (append (product-exp (car sos1) sos2)
                  (product (cdr sos1) sos2)))))

(define product-exp

```

```

(lambda (s sos2)
  (if (null? sos2)
      '()
      (cons (list s (car sos2))
            (product-exp s (cdr sos2))))))

```

Exercise 1.22

```

(define filter-in
  (lambda (pred lst)
    (cond ((null? lst) '())
          ((pred (car lst))
           (cons (car lst) (filter-in pred (cdr lst))))
          (else (filter-in pred (cdr lst))))))

```

Exercise 1.23

```

(define list-index
  (lambda (pred lst)
    (list-i pred lst 0)))

(define list-i
  (lambda (pred lst n)
    (cond ((null? lst) #f)
          ((pred (car lst)) n)
          (else (list-i pred (cdr lst) (+ n 1))))))

```

Exercise 1.24

```

(define every?
  (lambda (pred lst)
    (cond ((null? lst) #t)
          ((pred (car lst))
           (every? pred (cdr lst)))
          (else #f))))

```

Exercise 1.25

```

(define exists?
  (lambda (pred lst)
    (cond ((null? lst) #f)
          ((pred (car lst)) #t)
          (else (exists? pred (cdr lst))))))

```

Exercise 1.26

```

(define up
  (lambda (lst)
    (cond ((null? lst) '())
          ((not (pair? (car lst)))
           (cons (car lst) (up (cdr lst))))
          (else (append (car lst)
                        (up (cdr lst))))))

```

Exercise 1.27

```

(define flatten
  (lambda (slist)
    (if (null? slist)
        '()
        (append (flatten-in-s-exp (car slist))
                  (flatten (cdr slist))))))

(define flatten-in-s-exp
  (lambda (sexp)
    (if (symbol? sexp)
        (list sexp)
        (flatten sexp))))

```

Exercise 1.28

```

(define merge
  (lambda (loi1 loi2)
    (cond ((null? loi1) loi2)
          ((null? loi2) loi1)

```

```

      ((< (car loi1) (car loi2))
       (cons (car loi1) (merge (cdr loi1) loi2)))
      (else (cons (car loi2) (merge loi1 (cdr loi2)))))))))

```

Exercise 1.29

```

(define sort
  (lambda (loi)
    (if (null? loi)
        '()
        (cons (least-element loi)
                (sort (remove-first-occurrence (least-element loi)
                                                loi))))))

```

```

(define least-element
  (lambda (loi)
    (least (car loi) (cdr loi))))

```

```

(define least
  (lambda (l loi)
    (cond ((null? loi) l)
          ((< l (car loi)) (least l (cdr loi)))
          (else (least (car loi) (cdr loi))))))

```

```

(define remove-first-occurrence
  (lambda (x lst)
    (cond ((null? lst) '())
          ((eqv? x (car lst)) (cdr lst))
          (else (cons (car lst)
                      (remove-first-occurrence x (cdr lst))))))

```

Exercise 1.30

```

(define sort/predicate
  (lambda (pred loi)

```

```

(if (null? loi)
    '()
    (cons (first-element pred loi)
          (sort/predicate pred
                          (remove-first-occurrence
                           (first-element pred loi) loi))))))

(define first-element
  (lambda (pred loi)
    (find-first-element pred (car loi) (cdr loi))))

(define find-first-element
  (lambda (pred n loi)
    (cond ((null? loi) n)
          ((pred n (car loi))
           (find-first-element pred n (cdr loi)))
          (else (find-first-element pred (car loi) (cdr loi)))))

```

Exercise 1.31

Exercise 1.